4

AD-A229 053

RADC-TR-90-203, Vol I (of three)
Final Technical Report
September 1990

DOS DESIGN/APPLICATION TOOLS

Honeywell Corp.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RADC-TR-90-203, Vol I (of three) has been reviewed and is approved for publication.

APPROVED: [illegible]

THOMAS F. LAWRENCE
Project Engineer

APPROVED: [illegible]

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: [illegible]

IGOR G. PLONISCH
Directorate of Plans & Programs

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED Final Dec 87 - Dec 89

4. TITLE AND SUBTITLE
DOS DESIGN/APPLICATION TOOLS

5. FUNDING NUMBERS
C - F30602-87-C-0104
PE - 62702F
PR - 5581
TA - 21
WU - 78

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Honeywell Corporacion
Sensor and System Development Center
1000 Boone Ave, North
Golden Valley MN 55427

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COTD)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-203, Vol I (of three)

11. SUPPLEMENTARY NOTES
RADC Project Engineer: Thomas F. Lawrence/COTD/(315) 330-2158

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

[illegible]

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Developing applications for execution in a distributed processing environment is a difficult task. Such environments dominate Air Force C^3I systems, which are necessarily distributed. In addition to being a physical necessity, distributed systems offer, relative to centralized processing systems, the potential for increased performance and fault tolerance. Realizing that potential is a key objective behind research in distributed systems technology.

The goal of this contract is to:
1) Define and demonstrate a framework for integrating development tools, AfD
2) Define and construct tools that support the development of distributed applications.

A tool integration platform was designed and developed as a fundamental element of an integrated development framework. The RADC Distributed System Evaluation (DISE) Environment Tool Integration Platform integrates software development tools by automating and coordinating information exchange between tools, through use of the CRONUS distributed system and the ONTOS object oriented database management system.

(Continued)

14. SUBJECT TERMS
Software Development Tools, Resource Allocation, Tool Integration, Reliability Analysis, Distributed System, Object Oriented DBMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 880922
Prescribed by ANSI Std. 239 18
299-01

Block 13 (Continued)

Two development tools were selected and implemented that illustrate the types of technology required to support distributed application development. The Allocator assists developers with determining efficient implementations for distributed applications. The Reliability Analyzer generates reliability measures for application components given a set of hardware reliabilities. The two tools have been integrated into the IP.

This report summarizes the contract's objectives and results.

PREFACE

This is the Final Technical Report for RADC contract F30602-87-C-0104, *DOS Design Application Tools*. The contractor is Honeywell's Sensor and System Development Center. This report provides an overview of the contract's objectives, investigations and results.

Table of Contents

List of Figures

SECTION 1

Executive Summary

Developing applications for execution in a distributed processing environment is a difficult task. Such environments dominate Air Force C^3I systems, which are necessarily distributed. In addition to being a physical necessity, distributed systems offer, relative to centralized processing systems, the potential for increased performance and fault tolerance. Realizing that potential is a key objective behind research in distributed systems technology.

The goals of this contract are to:

1) define and demonstrate a framework for integrating development tools; and
2) define and construct tools that support the development of distributed applications.

An integrating framework provides principles and automation for harmonious cooperation among tools and uniformity of tool invocation and data access from the user's perspective. As part of the foundation for an integrated development framework, a tool integration platform was designed and developed under this effort. The DISE Tool Integration Platform integrates development tools by automating and coordinating information exchange between tools. Development tools of all kinds are readily integrated via this platform.

Two development tools were selected and implemented that illustrate the types of technology required to support distributed application development. The Allocator assists developers with determining efficient implementations for distributed applications. The Reliability Analyzer generates reliability measures for application components given a set of hardware reliabilities. The two tools have been integrated into the Integration Platform, thereby illustrating both the process of integrating tools as well as the benefits of using integrated tools.

This report summarizes the contract's objectives, investigations and results.

SECTION 2

Overview

This section provides an overview of the contract's goals, approach and results. Additional details regarding the approach may be found in the two interim technical reports noted below. The software specification and design documents should be consulted for technical details regarding the software developed under this effort. Detailed descriptions of the use and and maintenance of that software can be found in the programmers and users manuals that are also identified in that section.

Following this brief overview, Section 3 summarizes the results of the investigations that preceded the design and implementation of the Integration Platform and the two development tools. Sections 4 and 5 then discuss, respectively, the DISE Tool Integration Platform and the two development tools developed under this effort. Section 6 summarizes this report and discusses future directions.

2.1. Motivation

Distributed applications are inherently more difficult to develop than centralized applications. They are more difficult to design, debug and maintain due to a number of factors related to the physical distribution of processing resources:

- heterogeneity between processing resources;
- asynchrony between hardware platforms;
- typically large-sized applications;
- multiple types of interactions between application components;
- non-determinism of application execution.

Nevertheless, C^3I systems are, by their very nature, distributed processing environments. In addition, they offer the potential for increased performance and fault tolerance. Performance gains are possible simply because multiple computing resources are available. Increased fault tolerance can be realized since a single hardware failures do not necessarily result in total system shutdown.

It is therefore clear that technologies are required that assist developers deal with the inherent complexity of distributed application development and that help them realize the full potential offered in distributed computing environments. These technologies will take the form of development tools geared towards dealing with the problems faced in constructing distributed applications. Development tools are used by developers in the process of creating distributed applications. A distributed debugger is representative of the this class of tools: they allow the developer to analyze aspects of the application under development and assist with its the construction and implementation.

However, in the absence of software development mechanisms and policies that tie tools and activities together in a cohesive manner, such a tool set will result in a disfunctional system. It would consist of an ad-hoc collection of technologies in which it is difficult or impossible to leverage the benefits among tools in meaningful ways.

2.2. Objectives

The objectives of this contract are to:

1) Define and demonstrate a integrating framework suitable for distributed application development;
2) Define and implement tools that support the development of distributed applications and that illustrate the development framework.

2.3. Results

2.3.1. Integrating Framework

An integrating framework provides methods and facilities that coordinate the process of developing software. It is a set of mechanisms that supports the effective and appropriate use of development tools across the development life cycle. Integration assures easy transfer of information among tools, simplifies their use, facilitates consistency maintenance as applications are transformed from requirements to implementation and off-loads menial tasks from people to computers. Section 3.3 discusses further the motivation for and elements of integrating frameworks.

Tool integration takes on special significance in the context of the Distributed Systems Evaluation Environment (DISE), RADC's testbed for evaluating distributed system technology. DISE currently consists of a variety of heterogeneous computer systems, which are connected to the internet and running the Cronus distributed operating system software. Development tools in DISE are generally developed in isolation and therefore may not operate in a coordinated fashion. Furthermore, DISE is an evolving environment that will continue to acquire new, sophisticated development tools, such as instrumentation tools and performance analyzers, that need to be integrated into a cohesive technology base.

Therefore a tool integration platform, which forms the cornerstone of an integrating development framework, was selected for development under this contract. Tool integration mechanisms support and encourage cooperation among independently developed programming tools by automating data exchange and sharing between tools. The DISE Tool Integration Platform (IP) serves as an information repository through which tools can store and share information generated during development.

Tools can interface to the IP for all their data input/output requirements. The IP provides a canonical, well-defined and conceptually centralized database of tool input/output data entities. These entities are the various pieces that comprise the distributed application under development, such as source code modules, processing environment attributes, and design information. These entities are cast in an information model that characterizes the enterprise of developing distributed applications. The IP is a tool integration mechanism in the sense that it coordinates the interactions between tools; it simplifies the task of bringing a tool into the development environment and having it work with other tools in a coordinated fashion.

Section 4 presents an architectural overview of the DISE Tool Integration Platform.

2.3.2. Development Tools

During the investigation a collection of candidate tools were identified as providing important capabilities in a development framework for distributed applications. Two of these tools were selected for implementation on the basis that they are design-time tools, they deal with important problem areas in distributed application development, and they have diverse data requirements yet share some I/O data in common, thereby providing an effective demonstration of tool integration framework. Section 5 discusses the motivation behind the two tools and describes their features.

2.3.3. Delivered Software

Three primary software deliverables were generated under this effort:

Tool Integration Platform
: The Tool Integration Platform integrates tools in a distributed environment by automating data exchange between tools.

Allocator
: The Allocator determines an efficient allocation of application components to processing nodes in the distributed processing environment.

Reliability Analyzer
: The Reliability Analyzer computes expected availability of application components executing in a faulty processing environment.

The Integration Platform is a key element of an integrating framework for distributed application development. The IP facilitates the process of integrating new tools by providing a modular architecture coupled with run time services that effectively reduce the complexity of interfacing to the underlying database. The IP can be readily extended to meet the needs of DISE's evolving technology base, due to the expressiveness of the object model in which the underlying schema has been implemented. It provides a foundation upon which other framework elements can be built and offers immediate benefits to distributed application developers by coordinating interactions between tools.

The two development tools serve dual roles. They illustrate the type of technology needed by distributed application developers in the form of design-time tools. In addition, they provide a demonstration of the process and advantages of tool integration via the IP.

2.3.4. Delivered Documentation

The following documents were generated in the course of this effort. The reader is referred to these documents for additional detail and technical discussion.

First Interim Technical Report, July 14, 1988
: This report summarizes the work performed in the first 12 months of the contract, June 26, 1987 through June 26, 1988. The report investigates the problems faced by distributed application developers, discusses the notion of an integrating framework, defines the requirements for tool integration technology and and specifies a set of candidate distributed programming tools.

Second Interim Technical Report, June 21, 1989
: This report summarizes the work performed in the second 12 months of the contract, June 27, 1988 through June 26, 1989. It discusses issues in the design approach and implementation of the Tool Integration Platform (IP) and the two development tools. Included are the results of three studies, one presenting alternative database substrates for the IP, one analyzing alternative architectures for the IP, and one providing a long term vision for an integration framework in DISE.

System/Segment Specification, December 21, 1989
: This document defines software requirements for the IP and development tools.

Software Top Level Design Document, March 21, 1989
: This document provides the top level designs for the DISE Tool Integration Platform, the Allocator tool and the Reliability Analysis tool.

Software Programmers Manual, December 21, 1989
: This document describes how to maintain the delivered software. It has been partitioned into three parts. The first is the IP Administrator's Manual that describes how to maintain the IP.

The other two parts discuss how to maintain the development tools and IP/tool interface software.

Software Users Manual, December 21, 1989

This manual discusses how to use the development tools and provides a detailed description of how to integrate a tool via the IP.

SECTION 3

Investigations

The first year of this effort focused on investigations into the problems faced by distributed application developers and the types of support they need to deal with those problems. The results of these investigations appear in the *First Interim Technical Report* identified in the previous section. This section summarizes the results of these investigations.

This section begins with the study of distributed system characteristics as they relate to BMC^3 applications. The next study focuses on problem or *risk* areas that developers must face when building distributed applications. Following that is a discussion on integrating frameworks and their role in distributed application development. Finally, the last study identifies a set of representative tools that assist developers in dealing with the various risks of distributed application development.

3.1. Distributed Systems

This subsection outlines some of the key characteristics of distributed systems in general and BMC^3 systems in particular.

Distributed systems offer a number of positive characteristics not found in centralized, single-processor systems, including speed, flexibility and fault tolerance. Regarding speed, distributed systems offer the potential for linear increase in available computing power generated by connecting multiple, autonomous computing platforms. Distributed programs have several ways of utilizing this power to enhance their execution performance. Two logically concurrent program segments running on different hardware platforms can execute in parallel, thereby increasing execution performance relative to a centralized, inherently sequential configuration.

Alternatively, program execution can be enhanced through appropriate utilization of processing resources. A heterogeneous computing environment may offer differentiated processing services, for example one processor may have a large memory while another contains a floating-point coprocessor. By assigning a particular program segment to a processor which provides the types of services required by that segment, execution performance of the distributed program can be enhanced. In addition, through sensitive placement of segments, locality of data reference can be realized; this, too, can improve performance. The increase in computing power of distributed systems is of primary importance to computation-intensive and real-time applications.

Distributed systems provide increased computing flexibility. One way this is realized is through the use of differentiated processing resources, as noted above. In addition, distributed systems allow data to be located and processed where it is convenient to do so. For example, it may be possible to process sensor data or perform data compression where data is generated rather than transmitting raw data to a central processor. Incremental system growth, where new or foreign computing platforms can be added to the distributed system, also increases flexibility. Distributed systems enhance programming flexibility by providing a diversity of resource types and spatially distributed yet integrated computing platforms.

The final important advantage of distributed systems is in the area of fault-tolerant computing. Distributed systems offer the capability for continuous processing in the presence of site and communication failures. First, a distributed system provides graceful degradation; a single failure does not necessarily terminate activity at other sites in the system. Second, replication of data and function on different sites allows an application requiring a resource from a failed site to obtain it from an operational site. Redundancy of storage and processing resources can be exploited in the development of fault-tolerant applications to achieve higher availability and reliability.

Despite these advantages of distributed computing systems, there are a number of issues that must be addressed when developing applications to executed in these environments. In particular, BMC^3 applications tend to be large, complex, and expensive to develop and to maintain. These characteristics make it especially challenging for C^3 systems to meet their objectives in the distributed computing domain. Several unique features of distributed syste ccount for much of the problem.

Heterogeneity

Heterogeneity in distributed systems is a given. Existing investments in diverse hardware, operating systems and programming languages cannot be discarded. Furthermore, certain types of hardware platforms are better suited to certain tasks and less well suited for others, resulting in heterogeneity on the basis of required functionality. In C^3 applications in particular, it is quite possible that each of an application's components will be implemented on a hardware and/or software platform that best suits its needs.

Several issues are raised by heterogeneity, including naming, data representation, and communication. A component of an application on one computing platform must have a way to name a component on another, perhaps foreign, platform. This necessitates some acceptable system-wide naming conventions. The data representation issue arises since different machines represent data in different, often incompatible, ways. Some mechanism must be provided for coordinating data formats across machine boundaries. Finally, layered communication protocols must be able to accommodate heterogeneous platforms and networks.

Large-grained components

For execution in a distributed processing environment, a distributed program must be partitioned into a number of program units (objects and clients) and subsequently allocated to the various processors for execution. The granularity of the partitioning has to be small enough to exploit the parallelism offered by, and resources available across, multiple processors. But since each unit has a separate address space, the operating system overhead for handling units will become prohibitive if they are too small. And, units must be large enough so that it offers programming convenience and supports an appropriate notion of encapsulation of data or function; related functions, for example, should not necessarily be dispersed across object boundaries, thereby leading to unnecessarily high communication profiles.

The primary issues concerning large-grained segments are the partitioning and assignment problems. The partitioning problem addresses how to decompose a distributed program into an appropriate set of communicating units of distribution. This is a challenging design-time problem and can be viewed as a broader notion of the classic decomposition problems in centralized programs, where the question is how to decompose a problem into appropriate functions or modules.

The closely related assignment problem addresses how to assign these units to processors so as to achieve stated objectives. The assignment issue arises both at load-time (static assignment) and at run-time (dynamic scheduling). Partitioning and assignment have important ramifications with respect to a program meeting its performance and fault-tolerance objectives. Unfortunately, both problems are difficult to solve. In fact, the assignment problem has been shown to be inherently complex (i.e., NP-

Complete).

Real time requirements

Real-time processing requirements are pervasive in BMC^3 systems. Hard real-time requirements set both a deadline (upper bound) by which an action/event has to happen and a threshold (lower bound) before which the action/event should not happen. This requires sophisticated synchronization and scheduling schemes which must face the reality that optimal scheduling with such constraints is known to be an NP-hard problem. A particular challenge in this area is to engineer, at the language and operating system levels, scheduling and synchronization policies to minimize the time consumed by process switching and event handling.

Concurrency, asynchrony and non-determinism

A typical BMC^3 application consists of multiple objects that interact with one another asynchronously. The presence of parallel execution (from different processors running simultaneously and most likely at different speeds) and communication delays may cause events to occur in a non-deterministic manner. Concurrency and non-determinism make programs difficult to debug and verify since errors may not be reproducible and many events can be happening at once in the system. At design time, conceptualizing and utilizing the multiple threads of control available in a distributed program can be extremely difficult.

Failures

Fault tolerance is a very attractive capability provided in distributed systems, and is especially desirable for C^3 applications. Faults can occur in hardware, such as site and communication link failures, and in software. A hardware failure at one site should not, in general, terminate execution of the program at operational sites. Communication failures, especially where sites are separated by large distances which traverse diverse communication media, are simply a reality.

Replication, as we noted, is the key mechanism behind continuous processing in the presence of failures. However, several difficult issues still must be resolved in the development of fault-tolerant applications, such as concurrency control, maintaining consistency among replicated entities, and realizing atomicity of operations.

Communication

BMC^3 applications are expected to involve a great deal of communication. In fact, unlike those of LAN-based distributed applications in research environments, a typical BMC^3 application's components may be located several hundred miles apart. The cost of communication between program components on different hardware platforms is at least one or two orders of magnitude higher than a memory access. Therefore, communication protocols must be robust and efficient. At the language level, application developers require communication constructs which hide the underlying implementation complexities. However, a lack of sensitivity during application development and implementation with respect to communication costs can lead to severe performance degradation for the program.

Dynamic Reconfigurability

BMC^3 applications have a special requirement with respect to dynamic reconfiguration because of their possible location in hostile, distant environments and because of their evolving nature. Since these applications are expected to be maintained and enhanced even after deployment, mechanisms are required for upgrading components without bringing the whole system down.

3.2. Problems in Distributed Application Development

Distributed applications are more risky to develop than centralized, single-host applications for at least the following reasons: 1) they are inherently more complex than centralized systems; and 2) they are newer than centralized systems, so there is less communal and individual experience in developing them which leaves many development problems unsolved. A *risk* is an aspect of the system under development perceived as a particular threat to the task of meeting the system objectives. The threat is defined in terms of the potential consequences, i.e., possible deviation from the system objective, resulting from an inadequate treatment of the system aspect.

Risk identification is a subjective activity since perceptions are those of the development team. Risks may vary among problem domains, that is, risks are context dependent. Risks are discovered when considering alternatives at various stages of system development. Risks may also be articulated because a knowledgeable developer recognizes them from certain objectives ("10ms response time") or constraints ("use C").

A study of risk areas enables the identification and study of different resolution techniques in the form of development frameworks and tools. It also provides a basis for evaluating tools with respect to the tool's potential payoff in terms of risk resolution effectiveness

The following subsections present a variety of risk areas. Each risk area is defined in general and then in terms of individual development *risks*. These individual development risks are then abstracted into *areas of concern* for developers, and finally *resolution techniques* for each area of concern are identified. It is the resolution techniques that provide the basis for specifying high-payoff distributed application development tools.

3.2.1. Performance

Essentially all programmed systems must address performance objectives. Typically performance is defined as execution time, although a variety of other metrics are possible, ranging from memory and resource utilization to reliability and dollar cost. In fact performance can be viewed as one aspect of correctness, in the sense that a program running at 50 ms having an objective of 20 ms is not "correct." However, it is useful to distinguish performance and correctness by viewing performance as a measurable quantity that is closely related to system objectives yet which is independent of system functionality.

Execution time represents a convenient metric when discussing distributed program performance. In the context of distributed systems there are at least two important notions of time, *total time* and *response time* of the system. Total time is the sum of the execution times of all program units in the system. Response time is the maximum execution time across processors, where processor execution time equals the sum of execution times of resident units. Under either notion, execution time for a unit includes attendant communication costs experienced through interaction with other units. One of the key differences between these two notions is that response time decreases with increasing parallelism where as total time does not. In the following discussion we generally adopt the response time notion of execution time as the performance metric, acknowledging that other metrics might be more appropriate in a given situation.

Relative performance in centralized programs is primarily a function of the algorithm employed in the context of expected inputs. There is a single hardware platform and the compilation process (outside of some local optimizations) is a straightforward mapping from source program to executable code. In the case of distributed program development and implementation there are many other dimensions -- sources of risk -- which may have strong implications for performance. The sources of risk include algorithms, granularity of distribution units, synchronization structure, processor utilization, and communication costs. Performance is a risk area in distributed application development in the sense that achieving performance requirements may require that the developer manipulate these factors in a coordinated and timely fashion.

Risks

Algorithms

It is obvious that the particular algorithms developed for a distributed program will have a strong influence on runtime performance, just as with centralized programs. Furthermore, the existence of multiple, explicit threads of control, the additional level of abstraction (the unit of distribution), and mechanisms for their interaction (communication constructs) add to the complexities of distributed algorithm development and can have a significant affect on runtime performance through reduced parallelism and high communication costs. Hence, poor distributed algorithms can lead to a major degradation of program execution time.

Decomposition

A poor decomposition into units of distribution (objects too large, processes too small, etc.) may lead to inefficient patterns of execution. For instance, a decomposition which leads to a single unit of distribution eliminates the potential for parallelism and would tend to lead to poor resource utilization. On the other hand, a fine-grained decomposition migh' result in increased communication cost. Furthermore, the decomposition will affect the synchronization structure of the program, and a poorly synchronized program may exhibit increased response time by inhibiting parallelism.

Processor Utilization

Host computers in the distributed environment may vary with respect to the types of processing hardware offered. A unit of distribution may execute more efficiently on one type of host computer than another. In addition, multiple computers simply offer more raw computing power for the distributed application. An application which under-utilizes distributed processing resources, either through misuse or disuse, may exhibit increased execution time relative to an application which makes effective use of these resources.

Communication Costs

Communication, involving significant overhead due to underlying layers of protocols, buffer management, and process synchronization, is more expensive than a procedure call and is at least an order of magnitude more expensive than direct memory access. Therefore, as the only mechanism for distribution unit interaction, communication costs can have a significant impact on the execution time of a distributed program. Also, communication patterns leading to bottlenecks, within the communication network, at host processors, or at a particular process, can degrade response time relative to the case where the communication load is temporally or spatially spread out.

Areas of Concern

Distributed Algorithms

The manner in which distributed algorithms are developed will have a strong influence on program execution time. Distributed algorithms influence the pattern of decomposition, provide the basis for parallel execution, and dictate the patterns of communication between units of decomposition. Furthermore, the heterogeneity of hardware platforms may complicate distributed algorithm development. An algorithm's performance may vary depending upon the nature of the host processor where certain processors possess attributes (hardware support, memory, operating system) that predispose them to certain types of algorithms while making them relatively poor hosts to others. There may be no notion of a "best" algorithm independent of knowledge about the target host platform for each unit. The area of distributed algorithms is important concern for distributed application developers.

Partitioning

The problem of partitioning a distributed program into units of distribution is closely related to distributed algorithm development. They differ in that partitioning addresses the encapsulation issue in particular, whereas the distributed algorithms area deals with the internal mechanics of units.

By defining the number of distribution units and their sizes, partitioning influences program synchronization structure, sets limits on processors utilization, and influences the communication patterns in the system. Furthermore, a decomposition which isolates inherent parallelism into distinct units of distribution creates potential for parallel execution at runtime. Partitioning has implications for all the risks mentioned in the previous section.

Allocation

Given a collection of units generated by some partitioning technique, the allocation problem addresses how to allocate units to the underlying host processors so as to minimize execution time. The allocation scheme may influence program execution time in a variety of ways. High communication traffic between two units assigned to distinct processors may result in increased program execution time due to the relatively higher cost of remote communication. On the other hand, such an assignment creates the potential for parallel execution between the two units, thereby decreasing program response time. A particular allocation scheme may lead to communication bottlenecks at some processing nodes, and may affect the performance of algorithms in a heterogeneous environment, as noted earlier. It also influences the usage pattern of system resources (memory, ports, etc.), which may have a bearing on performance. Allocation is an important area of concern for program execution time through its affects on resource utilization, communication costs, and parallel execution.

Communication

Communication is a significant area of concern with respect to distributed program performance if for no other reason than communication is a relatively expensive operation. Communication also has implications for performance at the design stage. The particular semantics of communication mechanisms can affect the developer's model of unit interaction, thereby influencing how algorithms are derived and how units are partitioned into units of distribution. Furthermore, communication mechanisms may influence the degree of parallelism in programs, and hence affect response time, in that some may promote parallelism more than others. Communication is an important area of concern because all these factors play significant roles in determining the overall efficiency of a distributed program.

Resolution Techniques

Aspects of performance risk are distributed across the software development lifecycle. Decisions made from high level algorithm design through the final processor assignment step have major influences on the resulting execution time of the distributed program. Resolving the performance risk therefore requires techniques which can be applied at various stages of development.

Paradigms for Distributed Algorithms

Design and analysis of sequential, centralized algorithms is an active research field. However, automated tools for sequential algorithm development are currently lacking (although work continues in the areas of automatic and semi-automatic programming systems). The issues are only more complex for the development of distributed algorithms. The state of the art in parallel algorithm development has not progressed very far, with results oriented more towards development of fine-grained shared-memory parallel algorithms, where the target hardware environment is a multiprocessor, shared-memory platform as opposed to a multicomputer (non-shared memory) system. Since automated techniques are some distance away, for the near-term we can expect techniques for supporting the development of distributed algorithms to take the form of distributed programming *paradigms*.

The object-oriented approach to distributed programming has been used in the development of several distributed programming environments, including Cronus [Schant86a], HOPS [Silver88a], and Clouds [Dasgup88a]. While results are positive, some evidence suggests that the object-oriented paradigm does not adequately address a number of difficult issues, such as non-determinism and parallelism.

Another approach [Ramesh87a] suggests that distributed program development should begin with a centralized, shared memory version of the system that is semi-automatically transformed into a decentralized, message-based distributed version. Such approaches need further evaluation, as it is not clear that this technique leads to acceptably efficient solutions nor is it obvious that it simplifies program design.

Resolution of the partitioning area of concern may also be based on distributed programming paradigms. Partitioning is closely related to the problems associated with developing distributed algorithms, hence shares with it some of the inherent complexity with respect to automating resolution techniques. Object-oriented programming systems directly address the partitioning problem in promoting decomposition based around a data-server model.

Design Analysis

Developing a distributed algorithm and an associated program partitioning which meets performance objectives is a difficult task, even using suitable development paradigms. In any case, the extent to which the partitioning strategy actually leads to increased performance may depend on a variety of system attributes typically not available at design time, such as the nature of the processing environment (number of nodes, processing and memory capabilities), communication costs, allocation strategy, etc.

The capacity to conceptualize and analyze algorithms and decomposition alternatives could be enhanced by a graphical design tool. Such a tool would provide the ability to graphically represent units of partioning, attach expected workloads to those units, and annotate with expected patterns of interaction between units. Combined with a high-level model of the processing environment, this tool could support a variety of static and dynamic analyses of resource utilization, communication and computational bottlenecks, and concurrency anomalies in helping the programmer develop a reasonable, efficient decomposition. A design tool could provide assistance for dealing with all four areas of concern.

A variety of these types of design tools are described in the literature, and most are based on an underlying computational model. A Petri net is often used as the model, and provides an intuitively appealing interface in addition to well-known techniques for Petri Net analysis that can yield information about potential system bottlenecks and resource utilization. Other computational models have also been described, including generalized stochastic Petri nets, constrained expressions, and CSP-related models. Different models address different classes of design problems and provide different types of user interfaces.

Allocation Analysis

The allocation problem is closely related to the Multiprocessor Scheduling problem which, unfortunately, is known to be an NP-Complete problem [Garey79a]. Therefore, automated techniques which find the optimal allocation of program units to processors are not likely to be developed. In spite of this inherent complexity, many approaches to the allocation problem have been described in the literature. They fall into three basic classes: graph-theoretic, integer programming and heuristic. Most of these approaches could provide the basis for an automated allocation tool which assigns units to processors (perhaps suboptimally) so as to meet performance objectives. Such a tool would take as input descriptions of workloads for each unit, communication patterns between units, cost models of the host processors, and cost models of the communication system. From this information a system cost model can be derived which reflects both execution and communication costs for the program. It can then be employed under the particular search strategy to find a low-cost allocation of units to processors.

3.2.2. Failures

The increased number of components (i.e., machines, devices and communication links) in a distributed system increases the chances of a failure in the system (or decreases the mean time between failures). Guarding against the effects of these failures is one of the key issues in distributed computing systems. The treatment of software failures is an entirely different matter and is not treated here [Randel72a].

One of the ways of achieving *fault-tolerance* involves the use of protective redundancy. A system can be designed to be fault-tolerant by incorporating additional components and abnormal algorithms which attempt to ensure that occurrences of erroneous states do not result in system failures. Different degrees of fault-tolerance can be built into the system depending on the specifications, cost, purpose of the system etc.

Two kinds of failures are handled : site and network. Sites may crash and communications links may be interrupted. When a site crashes, it becomes temporarily or permanently inaccessible. Communication link failures result in lost messages; garbled and out-of-order messages can be detected (with high probability) and discarded. Transient communication failures may be hidden by lower-level protocols, but longer-lived failures can cause *partitions*, in which functioning sites are unable to communicate. A failure is detected when a site that has sent a message fails to receive a response after a certain duration. The absence of a response may indicate that the original message was lost, that the reply was lost, that the recipient crashed or simply that the recipient is slow to respond. However, *Byzantine* failures - failures that cause the processors/sites to behave maliciously - are assumed not to occur. Also, the processors or sites in the system fail independently of each other.

Risks

Unavailability of Data

If data is frequently unavailable or inaccessible because some site is down, users will perceive the system as unreliable.

Unprocessed Requests

The above mentioned problem of data unavailability may also result in user (or system) requests not being processed - just queued - till the required data becomes available, i.e., the site on which the data exists recovers.

Incomplete Processing

Actions - or transactions - may sometimes have to wait if a site fails during its execution. This is necessitated by the fact that the failed site may have contained data needed for the execution of this action - in this case, the problem becomes one of unavailability - or that the sites have to unanimously agree to either commit or abort the action to preserve consistency. Hence, the operational sites in the system have to wait - the activity is blocked - until the failed site(s) recover(s), and a consensus is reached.

Inconsistent Data

Preserving consistency in a single site case is a well understood problem. A *commit* is an unconditional guarantee to execute the action to completion and an *abort* is an unconditional guarantee to back out the action so that none of its effects persist If a failure occurs before the commit point is reached, then immediately upon recovery the site will abort the action.

However, in a distributed system, where one or more sites in the system may fail independently, the problem is more complex. Some sites may be ready to commit, and others ready to abort, but before a consensus is reached, a site may fail. To maintain data consistency, it has to be ensured that all sites participating in the action decide unanimously to abort or commit

Having multiple copies of data in the system introduces the possibility of data inconsistency among the various copies. It is the task of a *commit protocol* to enforce *global atomicity* and maintain data consistency and integrity.

Lost Updates

If a site fails after the data is updated but before it is written to some form of stable storage, that data or at least the updates made to it are lost.

Incorrect Results

The presence of inconsistent data and loss of data integrity may result in incorrect results/responses for requests.

Increased Response Time

The possibility of failures of sites at which requests originate, requests being blocked or just queued till a failed site recovers, etc., may result in increased response time.

Reduced Throughput

An increased response time would also result in a corresponding reduction in the throughput.

Areas of Concern

Availability

Availability is defined as the fraction of the time the system is operational over an observation period [Goyal88a]. A major concern is to provide continuous service the face of failures. Failures should be localized so that a program can perform its task as long as the particular nodes it needs to communicate with are functioning and reachable.

Data Consistency and Correctness

Data used and managed by the distributed system should be consistent in the presence of failures and concurrent activities This problem is even more acute in the presence of multiple copies of the data, where the possibility of the various copies becoming different over a course of time - thus destroying data integrity and consistency - is greater [Babaog87a].

In almost any system where on-line data is being read and manipulated by on-going activities, there are consistency constraints that must be maintained. Such constraints apply not only to individual pieces of data but to distributed sets of data as well.

Throughput and Response Time

Performance, as measured by response time and throughput, is an important area of concern. It may also call for different kinds of optimization algorithms to measure and improve performance in the presence of failures in the system [Patnai86a].

Reliability

Reliability can be defined as the probability that the *system* remains operational over an observation period [Goyal88a]. It is a measure of the success with which the system conforms to some authoritative specification of its behavior. When the behavior deviates from that which is specified for it, a *failure* is said to have occurred. Various formal measures of the reliability of a system can be based on the actual (or estimated) incidence of failures and their consequences : Mean Time Between

Failures, Mean Time To Repair, and so on [Randel78a].

Resolution Techniques

Replication

Replication is one approach to enhancing the availability of data on a site, in the context of site failures, when other sites in the system are operational. Hence, data and code can be stored redundantly at multiple sites to increase availability. Redundancy techniques can also be used to take advantage of the existence of multiple processors by duplicating processes on two or more machines. A particularly simple, but effective, technique is to provide every process with a backup process on a different processor. Thus if one process crashes because of a hardware fault, the other *cohort* process can continue.

When data and code are replicated on several sites in a system, failure of one or more sites on which copies of the replicated data are stored does not necessarily make that data unavailable to the rest of the system. An action requiring that data, can obtain it from one of the operational sites in the system that has a copy of the data. Replication, however, exacerbates the problems of maintaining data consistency, concurrency control, naming, addressing, recovery etc. Techniques to resolve/reduce the risks in some of these areas will be discussed later.

As replication of data provides copies of data at more than one site, the failure of a site containing such a copy does not necessarily involve blocking that request/action till that site recovers as the required data can be accessed at of the the other operational sites containing copies of that data. When copies of the data item are present redundantly on more than one site, any update to that data item requires a consensus from all the operational sites that have a copy of that data-item. Hence, if subsequently if one of those sites fails, the updated data is still available.

Replication may also improve performance. The possibility of increases in response time due to data unavailability is drastically reduced, as data availability is increased. Moreover, locality of reference, and proximity of data - maybe present locally, in which case, remote access is not necessary - may also result in improved response times. Similarly, improved response times, higher percentages of local data accesses, fewer blocked requests, reduced data location times etc., may result in higher throughput.

Failure Detection Protocols

The detection of failures in a distributed system in which the sites are subject to arbitrary failures is extremely important. To : (1) avoid unnecessary overhead in communication, queued requests, waiting for replies/acknowledgements from a failed site, etc., (2) minimize the problems of data consistency, concurrency control and (3) improve performance, failures should be detected as early as possible. The detection of the failure (recovery) of a site and the subsequent dissemination of this information to all the operational sites in the system - which results in this site being configured out of the system or into the system if it is a recovery - is handled by the failure detection protocols. This knowledge of the status of the system may prevent the possibility of user requests - and/or system actions - being queued, either waiting for a timeout or simply because the status of the site on which the action is to performed is not known [Chandy85a].

Hence, efficient failure detection (FD) protocols reduce the possibility of requests - that either originate from or require data available only on the failed site - being unprocessed and queued, by conveying the status of that site to the caller, and leaving it to the caller to decide whether to wait or just abort the action. Also, the early detection of failures in the system reduces the possibility of blocked unprocessed requests, reduces communication overhead thus reducing response times. Decreases in the response time of the system - due to reduced communication overhead - due to efficient FD protocols, and early detection of failures result in a corresponding increase in the throughput.

Atomicity

In the presence of replication, to realize fault tolerance, protocols are needed to provide coordination among the replicas in the presence of faulty processors. A widely employed control abstraction in distributed systems is that of an *atomic action* (atomic transaction) with the *failure atomicity* property : a computation structured as an atomic action can be aborted without producing any effects[Kohler81a]. An atomic action is an activity, possibly consisting of many steps performed by many different processors, that appears primitive and indivisible to any activity outside the atomic action. To other activities, an atomic action is like a primitive operation which transforms the state of the system without having any intermediate states.

Research in the areas of **reliability, recoverability** and **data integrity** has been concerned with the provision of atomic actions or transactions [Shriva86a]. Such an action is characterized by the serializable property : it is the unit of concurrency control such that concurrent execution of actions is equivalent to some serial order of their execution. For **reliability** purposes, it is also convenient to make it the unit of recovery [Traige82a].

Atomicity ensures that that the data managed by distributed programs are not rendered inconsistent by failures. Atomicity encompasses two properties : *indivisibility* and *recoverability*. Indivisibility means that the execution of one activity never appears to overlap the execution of another, while recoverability means that the overall effect of an activity is *all-or-nothing*, i.e., it either succeeds completely, or it has no effect.

The development of distributed systems supporting atomic actions - and recoverable objects in an object oriented distributed system - is an active area of research. Various programming language features have been proposed [Feldma79a]. By ensuring atomicity - i.e., guaranteeing the all-or-nothing property of all actions - the possibility of data inconsistencies and unresolved access conflicts affecting data integrity which may result in incorrect results is eliminated.

Non-blocking Commit and Termination Protocols

In the presence of failures in a distributed system if each site has a recovery strategy that provides atomicity at the local level, the problem becomes one of ensuring that the sites either unanimously commit or abort. A mixed decision results in inconsistent data. Commit protocols enforce such a consensus before a decision is made. Hence, commit protocols ensure atomicity. As mentioned earlier, atomicity encompasses indivisibility and recoverability. These two properties ensure that actions in the system are not terminated half-way, data integrity is not compromised and the effect of actions can always be recovered in the presence of failures, thus minimizing the possibility of incorrect results.

The indivisibility of the update action permits the developer to disregard the possibility of failures during the act of updating - i.e., writing to stable storage - and the recoverability property of atomic actions means that if the site fails before the update, the update can be recovered after the site recovers. Operational sites sometimes have to wait on on the recovery of failed sites to continue processing, if a site fails in the middle of an atomic activity. A protocol that never requires operational sites to block until a failed site has recovered is a *non-blocking protocol.*

Concurrency Control and Replicated Copy Control Algorithms

When data is accessed concurrently by more than one action, some form of concurrency control has to be exercised to maintain consistency of the data and prevent and resolve access conflicts. Several algorithms have been proposed to implement concurrency control in distributed systems, for example two-phase locking, consensus locking, version voting and time-stamping [Stoneb79a]. One of main issues in handling replicated data in distributed systems is to maintain data consistency. This is achieved by replicated copy or consistency control protocols. These methods can be classified into two broad categories : *pessimistic (conservative, blocking)* and *optimistic (non-blocking)*. Examples of conservative CC methods are voting schemes, primary copy methods and token-passing schemes.

Optimistic CC methods do not seek to ensure global consistency of replicated data during failures and network partitions. Thus access is always granted, but all conflicts are resolved during the commit or merge phase by the use of back-outs or compensatory actions. Note that tradeoffs can be made in this case between availability and consistency. Concurrency control algorithms attempt to (1) resolve conflicts that result when concurrent activities try to access data elements - to either read and/or update them, and (2) improve concurrency by allowing as many actions to interleave and execute as possible without violating the atomicity constraints. These algorithms hence try to ensure correctness - at least as it pertains to the effect of an action and its result.

Response time increases as the number of actions/requests handled by the system concurrently increases. Hence, efficient CC mechanisms that permit more tasks to execute concurrently increase RT. Finally, efficient CC algorithms allow a greater degree of interleaving of actions and increase throughput.

Logging/Checkpointing

Logging and checkpointing are techniques which allow the state of a site/object/data-item to be recorded on stable storage at frequent intervals. Hence, if this is done either at regular intervals, or at the start/completion of certain events, it is possible to go back, read the state of the system as it was - a short while - before the failure and *recover* the site.

3.2.3. Security

Risks

Some of the *security* risks in a distributed system are:

Unauthorized listening

Forging a principal's identity

Tampering with messages

Masquerading as somebody else

Gaining access to the system after the real user logs in

Forging signatures

Efficiency

Location and time varying physical security environments

Factors related to jurisdiction

Interactions between different access control implementations/mechanisms

Areas of Concern

Secure communication

In most distributed computing/processing systems, there are a lot of activities that involve communication. Most of the networks - and internetworks - used in such distributed systems are *open* in the sense that they are readily vulnerable to eavesdropping and interference from unauthorized *intruders*. Such an architecture presents security problems much different from the ones faced in centralized single-site systems. In particular, it is clear that security must be based on the use of *encryption* in the communication protocols. Encryption permits the establishment of a data channel that is less *open* than the underlying network by arranging that only authorized parties can create, inspect and/or modify some or all of the data.

Establishing, using and maintaining such a secure data channel requires the resolution of several problems. First, it is necessary to identify the authorized parties (traditionally called *principals*). Second, it is necessary to convince each principal that the others are indeed who they claim to be (usually called authentication). Third, it is necessary to transfer the actual data in a manner that is not vulnerable to various known threats.

Authentication

Authentication means verifying the identity of the communication principals to one another. In a distributed system in which a large number of computers communicate, and have no central machine or system that contains authoritative descriptions of the connected computers etc., there is need for decentralized authentication. Three kinds of communication can be considered here:

1) Authenticated interactive communication between two principals, i.e., a 2-way communication.
2) Authenticated one-way communication, such as is found in mail systems, where the sender and recipient may not be simultaneously known.
3) Signed communication in which the origin of a communication and the integrity of the content can be authenticated to a third party.

Multi-level security

The basic multi-level security policy is based on assigning different levels of access/security clearance to each individual/user of the system, with the fundamental requirement that no individual should see information classified *above* his clearance. This policy is enhanced by the use of *compartments* or categories designed to enforce need-to-know controls on the sharing of information.

Resolution techniques

Central authentication service/manager

Central authentication supports logon in a distributed system without requiring each user to remember more than one password. A special distributed node called an *authentication manager* manages user passwords and authenticates user requests [Needha78a].

Resource server

A strategy for distributed discretionary and mandatory access control utilizes the concept of an object manager or a *resource server*. This approach supports applications that are awkward to implement as a set of interactive terminal sessions. Each distributed system resource (i.e., object) is managed by a resource server that mediates access to it. Access control can be based on either access lists or capabilities.

Encryption

Secure communication in physically vulnerable distributed systems depends upon *encryption* of material passed between machines. It can be assumed that it is feasible for each machine in the system to encrypt and decrypt material efficiently - and moreover, with arbitrary *keys* that are not readily discoverable by exhaustive search or *cryptanalysis* [Chaum85a]. It can also be assumed that the computing environments themselves are safe/secure, as only secure *communication* issues are discussed here. The intruder(s) can interpose a computer in all communication paths, alter or copy messages, replay messages and emit false material.

In the presence of a central authentication manager/service, the principals that want to communicate with each other can negotiate with the authentication service to obtain a shared *conversation key*. This key can then be used to encrypt and decrypt subsequent communication between these sites.

It is possible to include secure communication at various levels in a communication protocol hierarchy. At the physical layer, security can be achieved by various non-cryptographic techniques that prevent tampering with the communication medium itself. At the data-link layer, it is possible encrypt all traffic on each link using a code whose key is shared among all nodes directly connected to that link. This is called *link* encryption; it protects against others from outside the community that shares the data link, but does not distinguish principals within that community. However, in networks that have multiple links, link encryption allows intrusion by members of the trusted community of *every* data link traversed by that path. The lowest layer at which end-to-end guarantee can be provided is the network layer, where direct node-to-node addressing of packets can be used.

Collection of subsystems with different security levels

Distributed systems can be thought of as a collection of subsystems (zones) with different facilities and security levels. Depending on the zone/level, the system can enroll new users, serve as a publicity medium and help identify illicit users [Bottin86a].

Negative security zone

This would be a part of the system that the system owner *wants* a stranger to access. It must not be possible for the user to leave this zone except by logging out. One reason for wanting users to see this data is that they may contain useful advertisements and/or advice. Or, it may be a *hacker trap* against an improper next level access. It may be a place that entertains the illicit user for a while until the call/user can be traced.

High-security zone

In this zone, the user must be positively identified. Any of the classic non-automated identification by personal information technique can be used. The system could keep an encrypted dossier on each of the users and ask a random set of questions to validate the user's identity. This makes it difficult to break even by eavesdropping, and moreover, mean time to break can be made arbitrarily large by increasing the number of questions asked and the size of the dossier.

Neutral security zone

This zone can be designed to encourage and identify new users. When a user attempts to log in and is not recognized, the system asks for information and constructs his dossier. This dossier can be verified, and the new user then allowed access to the high security zone.

Separation

One approach to the design of secure multi-level security systems is based on the key notion of separation. Four different separation mechanisms can be used : physical, temporal, logical and cryptographic. Physical separation is achieved by allocating physically different resources to each security partition and function. Moreover, distributed systems are well suited for the implementation of such systems.

Temporal separation allows untrusted host machines to be used for activities in different security partitions by separating those activities in time. The system state will, however, have to be reinitialized between activities belonging to different security partitions.

Logical separation can be provided by having special purpose kernels whose only function is to provide separation.

Cryptographic separation uses encryption and related techniques to separate different uses of shared communications and storage media.

3.2.4. Communication

The programs of a distributed computing system operate in an integrated fashion in order to achieve the common system goal. One of the unique characteristics of distributed systems is that there is usually considerable communication among units of distribution. Units must communicate to exchange pertinent information. Geographically dispersed processing units rely on the communication medium to achieve this goal. The underlying communication subsystem provides the facilities to exchange messages between sites. However, the existence of the subsystem also introduces several additional risks to the development of distributed software.

Risks

Communication Link failures

Communication link failures result in lost, garbled and out-of-order messages.

Communication Delay

The delays associated with communication activity often cause events to happen in a non-deterministic manner. Non-determinism makes programs difficult to debug and to prove correct. Moreover, delays caused by communication may affect the performance of the system. As units may interact with one another asynchronously, asynchrony is also a problem.

Incompatibility of Data Representations

In a distributed environment, the internal representation of data may be different on different hardware platforms (e.g., a byte-swapped integer representation on a particular node and something else on other node). Data exchanged between sites must somehow be converted. Without suitable software support, distributed application developers must know all the different representations and code each of the conversions manually. This not only places an unreasonable huge burden on the developers but may also increase the possibility for conversion errors.

High Implementation Effort

Implementing communication among distributed modules increases the effort required because there are more system dimensions to consider. Naming and addressing mechanisms are required to be able to locate and communicate with remote resources. The developer may have to use communication protocols to write higher level flow control facilities and make system calls to the underlying communication subsystem.

Areas of Concern

Reliability of Communication Medium

Communication media are subject to failures. Messages can be lost, garbled and out of order due to communication link failures. Transmission errors can also result from a variety of physical events [Tanenb81a] such as power surges. Errors can also be introduced whenever the receiver gets out of sync with the transmitter. Furthermore, the network might become partitioned, making it impossible for some nodes to communicate with each others.

Performance Degradation

Communicating overhead can cause performance degradation. The delay could be due to the time needed to process messages, queuing delay and propagation delay over long distance. Overall performance degradation may result from communication link failures.

Data Incompatibility

One of the concerns in developing distributed software is the difference in data representation schemes on the different hardware platforms used. All data transmitted across the network must be made compatible when communicating in a heterogeneous system.

Level of Language Support

Without suitable language support, distributed application developers must deal with the specifics of the communication layers within the system. The complexity of having to deal with every aspect of the communication can be quite overwhelming. Abstraction is needed to reduce the complexity of building distributed applications and free developers to concentrate on the application itself rather on the specifics of the distributed environment.

Resolution Techniques

Reliable Communication Protocols

Transmission lines are often subject to failures, causing errors. Use of more reliable medium is one way to improve reliability. Communication protocols at various levels are needed to provide reliability. Transient communication failures may be hidden by lower-level error detecting/correcting codes. Garbled messages can be detected with high probability and discarded. Error control protocols can either discard all messages other than the next one in sequence or buffer out-of-order ones until they are needed.

Efficient Protocols/Implementations

The efficiency of a communication subsystem is determined by many factors including some statistical characteristics of the transmission lines. Certain implementations such as broadcast directly at the hardware level can greatly reduce the overhead/delay incurred when the same implementation done in software. To achieve better efficiency, lower-level variables such as the timeout interval, sender/receiver window sizes and the frame size, along with many others can be fine tuned experimentally. Among the main issues in designing protocols above the layers of the communication subnet are naming and addressing, connection establishment and termination, flow control, buffering, multiplexing and error recovery. Depending on the assumption of the reliability of the subnet, virtual circuit and/or datagram connections may be preferred for the transport service.

Canonical Data Representation

Data transmitted over the network may have to be encoded and decoded at the sending and receiving nodes if the nodes use different representations for the data types in a message. To facilitate this conversion, a network canonical form to encode and decode any system's representation is preferred to a pair-wise conversion. For example, the eXternal Data Representation (XDR) [Micros86a] published by the Sun Microsystems for translating data and their corresponding external representations is one of such canonical representation. The XDR standard depends on the assumptions that bytes are portable and that the bytes' meaning across hardware boundaries would be preserved. The concern of incompatible data representation can be addressed by adopting a standard which is independent of languages, operating systems and hardware architectures.

Sufficient High Level Language Support

Units of distribution that communicate over a network need a paradigm for communication. A low-level mechanism might send a signal on the arrival of incoming packets, causing a network signal handler to execute. A high level mechanism could be language construct such as Ada "rendezvous." Providing language support that masks the network complexity can ease the task of writing distributed software considerably. High level communication constructs may be built based on a remote procedure call paradigm, asynchronous or synchronous message passing, broadcast, or any combination. Any of these communication mechanisms might be appropriate as long as it has well-defined semantics and provides location transparency.

3.2.5. Correctness

When a system meets its specification and/or performs as expected, the system is said to be correct. A system under development is at the risk of being incorrect throughout the entire development process since deviation might be introduced at various stages. The risk of correctness exists in centralized programming, however, it is confounded by the inherent complexity of the distributed computing. Establishing correctness of distributed programs is extremely difficult. To reduce or resolve the risk, we must greatly enhance our capabilities in verification and validation of both the design and implementation.

Risks

Incorrect Design

There are many more system dimensions to consider in developing a distributed software than in developing sequential programs. Distributed systems have a tendency to be "large", so we can expect that the design process to be more arduous. Other considerations such as communication protocols must be taken into account as early as design time. Also, the nature of the underlying hardware affects the design much earlier in a distributed environment. Since it is more complex to design distributed software, it is more likely to make errors in the design.

Incorrect Implementation

After the distributed software is written, other than the fact that it tends to be large, it is harder to detect and to remove errors in implementation.

Inconsistent Data

In the presence of multiple copies of data in a distributed system, data inconsistency is an acute problem. Various copies of data may become different over a course of time. The use of incorrect data will cause the system to deviate from its expected behavior.

Concurrency Anomalies

An incorrectly implemented systems may result in all kinds of concurrency anomalies such as deadlocks or other race conditions.

Areas of Concern

Design

Designing distributed software is much more complex and needs automated tools to simplify the task. Errors introduced at design time become costly to fix at implementation time. The ability to verify and validate a design is essential for the successful development of correct distributed systems.

Debugging

Dbx-style debugging of implementation is difficult since program parts are distributed. Even if such tools exist, debugging methods applied at the level of computational units in general cannot provide the high level viewpoint necessary for dealing with all the complexities of a distributed system [Bates82a]. The major difficulties in debugging distributed software are:

(1) Non-determinism

Debugging distributed software is tougher due to the non-deterministic and distributed nature of the control threads. Multiple threads of control make the program harder to understand and more prone to bugs.

(2) Definition of program state

Critical information regarding the bug might be lost due to the ongoing execution of other processes, communication delays, and local operating system functions. This makes it difficult to define and capture the entire state of a distributed program at any given instant.

(3) Reproducibility

The results observed in the distributed system are in many cases hard to reproduce. This is because of the nondeterministic nature of multiple asynchronous processes. The results do not only depend on the system input, but also depend on the relative timing of the activity [Garcia84a].

(4) Debugger Interference

In a distributed system, the processing delays caused by the debugging activity itself may make certain synchronization errors appear or disappear. If the debugging can slow down the execution of some process, it can influence the significance of the execution. The interaction between debugging activity and the software being debugged must be addressed.

Resolution Techniques

Formal Proof Techniques

Testing can be performed to uncover errors, but it cannot be used to prove program correctness. Correctness properties of a design or implementation can be validated by formal proof techniques. Correctness properties usually deal with state spaces, communication questions such as deadlock, starvation, or flooding, and reachability. The goal of correctness proofs is to find errors that cause predicates to be false and remove them by amending the text of a design or implementation [Pressm83a]. Proofs of program correctness span a broad spectrum of sophistication. Manual correctness proofs, such as the use of mathematical induction or the predicate calculus are suitable only in the evaluation of small programs To validate larger software systems requires automated proof technique. Some work has been done in this area but much work remains to be done towards an easily applied and validated method for formally proving the correctness of distributed software .

Static Analysis Tools

Static analysis tools can be used to support examination of static allegations- weak statements about a program's structure and format. Static analysis of code such as the "worst case" scenario and state-space analysis for concurrency anomalies such as deadlocks and starvation can be performed. These types of tools usually require user-guided distillation.

Automatic Programming Tools

Automated tools for applying correctness-preserving transformation for a program in one representation to another (e.g., design to program stubs) can minimize the chance for errors which might be introduced through manual transformations.

Design Analysis Tools

Design analysis tools that will establish certain program properties using the underlying framework for developing distributed software are of great help. Distributed design tools for visualizing program states or control can aid in dealing with the complexity problem.

Execution Monitoring

A monitoring system which collects, interprets and displays information concerning the interactions among processes will support both debugging at interprocess level and also performance evaluation. Timing studies are a desirable feature of the monitor. The interactions among executing processes can be monitored in terms of interprocess communication or other significant events (e.g., process creation, message transmission). The correctness of such program depends on the contents and sequence of messages transmitted between processes.

Interactive Distributed Debugging

The majority of source-level debuggers presently available are for debugging sequential programs. There are prototype systems such as DEFENCE [Weber83a] which has special features to assist the user in understanding the interactions of the executing processes. Through these features, the developers of distributed software can not only determine the next line of execution for each process but also control the invocation of processes. An interactive distributed debugger facility will also offer the ability to examine and modify variables, to set trace areas, to set conditional breakpoints and to invoke source-level step-wise execution.

Instrumentation Tools

In a distributed system, instrumentation can be used for debugging purpose. The performance of the distributed system can be measured against its requirements and bottlenecks can be located by determining where the majority of elapsed time is spent during certain operations [Bhatt88a].

3.3. Integrating Framework

This subsection motivates the notion of a framework for integrating tools for distributed applications development and introduces the framework elements.

3.3.1. Overview

It is well known [Penedo88a] that automated development tools in an integrated environment ease developers' burdens and lead to the production of more reliable software. Integration assures easy transfer of information among tools, maintains consistency of information as the application under development is transformed from requirements to implementation, provides a uniform interface for user interactions with tools, and off-loads menial tasks from people to computers. CASE technology for developing centralized information processing systems is already available on PCs at almost nominal cost [Chikof88a]. The benefits of integration should also be available to developers of distributed systems.

One of the objectives of this program is to provide a framework for integrating the tools in RADC's evolving DISE environment. The framework should support the most effective application of the DISE tools, present and future, planned and as yet unknown. Our integration technology applies to more than DISE, but it will be tailored to the requirements of DISE.

There are two major influences on our framework: the distributed nature of the software whose development it will support, and the fact that it is intended for the DISE environment.

Distributed systems pose development risks for two reasons:

- They are inherently more complex than centralized systems, therefore they are simply harder to develop no matter how much automated support is available.
- They are newer than centralized systems; there is less communal and individual experience in developing them. This lack of experience leaves many development problems unsolved that will be solved in the next decades.

The framework elements and development tools they support must all contribute to mitigating the risks of developing distributed software. The integrating technology we provide will be used first in RADC's internal DISE research environment. DISE's goals [Newton87a] are to

- provide an environment for the design and development of distributed systems,
- serve as a running demonstration of the systems integration technology within which distributed systems will play a part,
- provide researchers with the tools necessary to evaluate current distributed systems technology that is integrated into a distributed application environment suitable for experimentation,
- provide AF C2 systems with the capability for "system evolution," thus negating the need for total system replacement that currently exists.

To meet these goals, we envision DISE containing conventional development tools such as compilers, editors, linkers, a configuration management system and the like, and tools for experimenting with applications under development, tools such as interactive debuggers, monitors, performance measurement tools or simulators. Also DISE will evolve and that tools that aren't conceived of today will be inserted into it. The integrating framework must accept sophisticated unknown tools that support experiments with software under development as well as conventional development activities. It cannot overly constrain the tools it will integrate; for example, it cannot require that they are language specific.

A software engineering environment (SEE) supports software development with:

- a software process that control and disciplines development activities,
- high level, software-development-specific tools and capabilities that are implemented using lower level capabilities such as an operating system or a project database.

An *integrated* SEE (ISEE) supports harmonious cooperation among tools and exhibits certain uniformities from a user's point of view. The complete ISEE (Figure 1) provides:

- a technical and management software process, tailorable on a per project basis,
- automated guidance for or enforcement of both the project management and the technical processes,
- methods for accomplishing every activity in the management and technical processes, and tool support for the methods,
- a uniform user interface for all interactions between users and the SEE,
- automated data exchange and sharing among all project management and technical tools,
- automated capture and retention of all information used by any tool or person during develop-

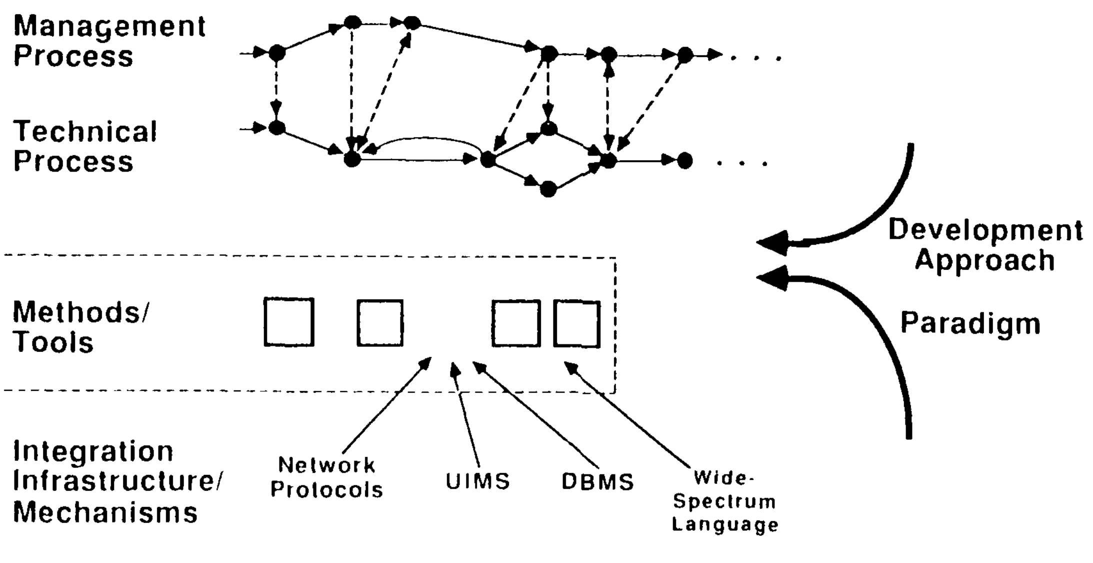


Figure 1. Vision of an Integrating Framework

ment,

- extensibility that permits new tool types and/or data types to be introduced into the environment during a single development project.

Integration is achieved with the software processes (which bring coherence to the collection of development activities), the uniform user interface and automated data exchange and sharing. An ISEE's *integrating framework* provides principles and automation to implement intergration.

An integrating framework provides principles and automation to implement harmonious cooperation among tools and uniformity of tool invocation and data access from a user's point of view.

Firstly, the framework provides a software process that implements a project management and technical strategy and disciplines the use of technical tools. It is not a piece of software, although it can be encouraged or enforced by tools that are pieces of software. Software engineering does not occur without a process. The process integrates tools in the sense that it insure a synergistic combination of methods and tools.

Automation for integration consists of an infrastructure (e.g., project DB) and mechanisms to cover what the infrastructure does not. The line between infrastructure and mechanisms is fuzzy at best. A rule of thumb is that the infrastructure provides support for tools that are built on top of it while mechanisms operate at the same level as tools. This distinction is important only because an ISEE should provide some infrastructure that implements an integration strategy, not just a collection of mechanisms.

Infrastructure elements are such things as:

- a wide spectrum language containing all the concepts and syntax needed to formulate different versions of a program – requirements, designs, implementations,
- a project data model and a database that holds all versions of a program and all other technical and management information needed to specify, design, implement and document the program and control the development process,
- a user interface management system that support the development, execution and maintenance of user interfaces for all the tools in the ISEE and for the ISEE itself,
- operating system level interfaces such as CAIS or X-windows,

Integration mechanisms deliver the same benefits as infrastructure elements, but they are more *ad hoc*. For example, data transformers allow tools to share data objects even though each has a different view of the data's format, and shell scripts allow users to invoke several tools using similar protocols even though each tool expects a different invocation protocol.

A particular ISEE framework provides as many infrastructure elements and integrating mechanisms as needed to achieve the level and kind of integration the ISEE proposes to provide.

An ISEE for developing distributed software contains tools not found in an ISEE for centralized software but does not contain a substantially different software process or integrating framework. If, in addition, the distributed software must evolve over time without halting the system, languages and run time systems must support dynamism. But, again, the software process and integrating framework for environments to develop dynamic distributed software will not be substantially different from those used when developing centralized software.

If, however, both the software and the development effort are geographically distributed, the ISEE must offer a software process (especially a management process) that can account for people and software modules not all being in the same place, a distributed integrating framework, and development tools that operate in a distributed environment.

Ultimately, an ISEE should not be an environment instance but a meta-environment, a generator which produces tailored environment instances on a per project basis. In addition to the usual integration framework and development tools, such an environment will contain tool building tools and support for tailoring. The subject of environment generation is outside the scope of this project.

During the investigations, two framework elements in particular were identified as having highest initial payoff. They provide a basis upon which other integrating framework elements can be based. Therefore, this project proposes an integrating framework that provides:

- a technical software process that is in reality a meta process that project members will tailor as they progress through the steps in the project. This is a compromise between support for only a single process (e.g., IDE or Excelerator) and automated tailoring upon environment instantiation for an individual project.
- automated data exchange and sharing among technical tools *via* a project database that, for the moment, hold technical information but that is extensible.

3.3.2. Software Development Process – Risk Driven

3.3.2.1. Overview

A *software process*, often called a software development lifecycle, is the partially ordered collection of related activities (e.g., design, implementation, testing, integration) involved in producing a software system. Articulating a process orders and organizes these activities into a coherent flow subject to reasoning. A software process is an integrating mechanism because it defines and orders development tasks and suggests appropriate methods and tools for each task and for the transitions among tasks.

Decades of experience in developing centralized software has established that using a software engineering process greatly improves the chances of attaining the desired system within resource constraints. Software processes are part of the integrating framework; they define the development tasks, control the order among them, and suggest appropriate methods and tools for each task and for the transitions among tasks.

Software development processes are defined with drivers in mind. For example, the major driver behind the waterfall lifecycle [Boehm83a] is the generation of work products: requirements, designs, source code, object code, documentation and so forth. Therefore the tasks it prescribes, the methods and tools for accomplishing those tasks, and the order among tasks are aimed at producing work products.

A software process has two major components:

- a *technical process* (also called a methodology) which dictates the technical activities used to develop a software system and a partial ordering among them. Technical activities include design, coding, performance analysis and so forth.
- a *management process* which dictates the activities used to control (order, manage and measure) the technical process and to measure some attributes (e.g., quality) of the product. Management activities include requirements tracking, design analysis for maintainability, assuring that documentation and code remain in synch and so forth.

Some software processes prescribe what activities should be done and the problem solving techniques to use in doing them. Many others define the activities and the ordering among them but not the problem solving methods used in doing the activities.

A software process has both technical and management aspects. A software process definition may consider only the technical process or only the management process. When the definition defines only one aspect of the "process in action," developers should define the other aspect of a complete software process before starting development.

A *method* or *technique* is a problem solving recipe for accomplishing an activity. For example, data flow analysis and functional decomposition are two technical methods for doing software design. A *development approach* is the philosophy behind, or motivation for, a software process. A process puts the philosophy into practice by guiding the application of problem solving methods and, to some extent, influencing what methods are chosen. For example, a top down decomposition approach leads to a classical waterfall development process while the build and evaluate approach leads to an incremental development process. *Software tools* automate technical methods.

3.3.2.2. The Spiral Meta-Process

Developing distributed systems is inherently more *risky* than developing centralized systems because there are more system aspects to consider and control (distribution, allocation, fault tolerance, performance) and fewer technical methods for considering and controlling them. Therefore we advocate a risk driven development approach. That approach of considering and resolving risks will determine the development process.

The spiral model of software development (Figure 2) implements a *risk-driven* rather than document or milestone driven approach [Boehm86a]. Recognizing, investigating and resolving risks are explicitly planned development activities. The spiral process includes most previous processes (waterfall, two-legged model, evolutionary development models) as special cases. It holds that software is developed in cycles,

> "...each cycle involving a progression through the same sequence of steps for each portion of the product and for each of its levels of elaboration [Boehm86a].

At each cycle, risks, or uncertainties, can be resolved by prototyping, simulation, analysis or even past experience. The major strength of the spiral process is that it controls development formally with the same driver – risk – that controls it *in fact*. Its major weakness is its newness; there is little experience with using it.

Most software development processes prescribe what technical activities should be applied in what order to produce a system. For example, the waterfall process says: do requirements, then design, then implementation. Some processes may even prescribe how to accomplish an activity: e.g., do design by functional decomposition. The spiral process prescribes neither what technical activities should be done nor the technical methods (problem solving techniques) for accomplishing the activities. It defines the driver for development – risk – and guides the development activities by prescribing that each of them is done in a four stage cycle. For this reason, the spiral process is, in fact, a *meta-process* or *process template* rather than a software process like the waterfall.

All development approaches articulate philosophies. Development processes put the philosophy into practice by guiding the application of problem solving methods and, to some extent, influencing what methods are chosen. The spiral software development meta-process is particularly well suited for distributed development because it simply guides method application and does not, in fact, prescribe what task (design, implementation, testing) is done in each cycle, nor the problem solving technique for the task, but only that the task is done as a four phase cycle. This makes the spiral a meta-process, a tool for defining processes, a template into which some software development process or mix of processes is inserted. As cycles are accomplished, the process or mix of processes to be used becomes clear from risk identification and resolution.

It is probably true that a traditional software development process (waterfall, prototyping, incremental development) can be selected before development starts and carried out as a number of four stage cycles. But a risk driven development approach works best when it retains the flexibility of choosing the next activity based on identified risks and their resolution.

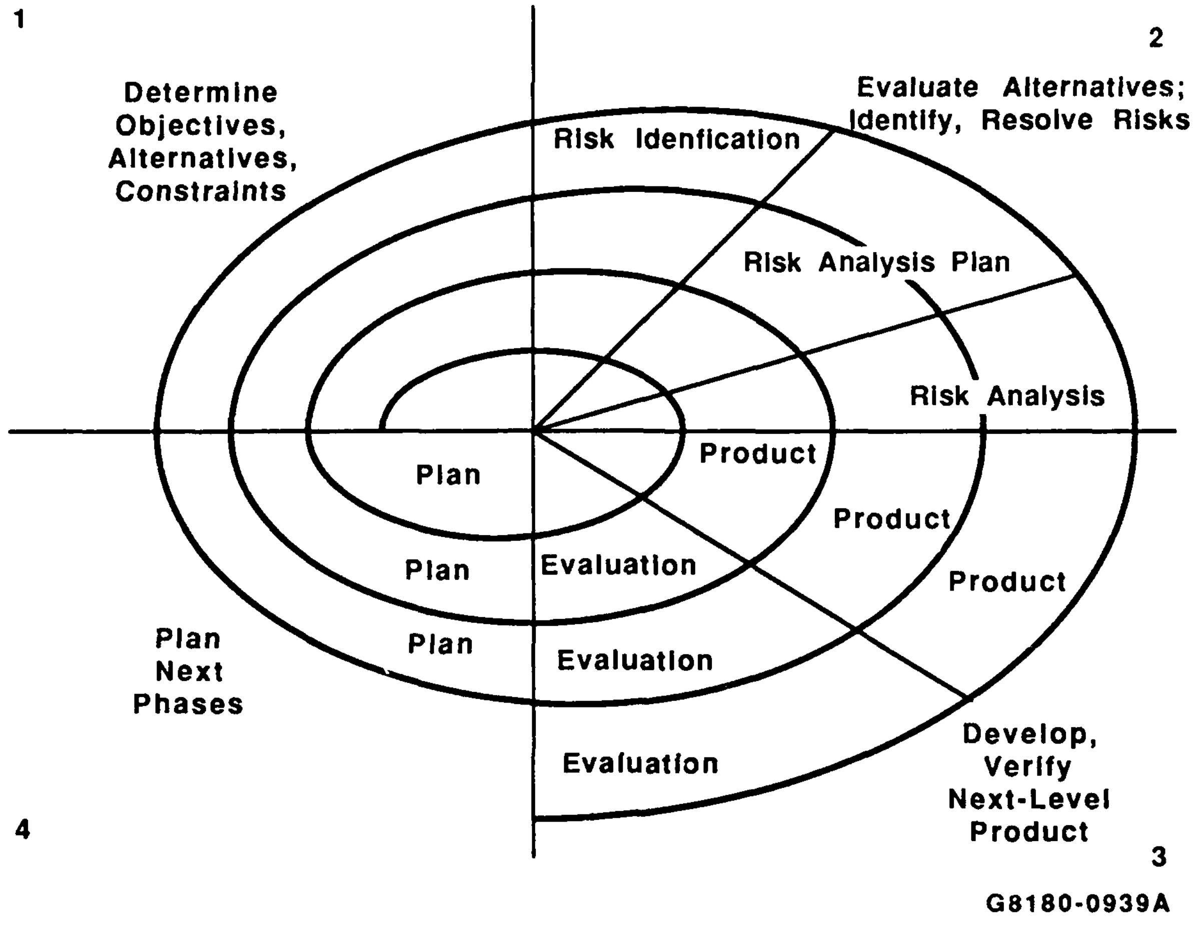


Figure 2. Spiral Software Meta-Process

The fact that the spiral is a meta-process causes it not to take different forms for developing centralized and distributed systems. Instantiations of the spiral will differ in what tasks are done to accomplish an activity such as design or implementation. For example, allocation of objects to processors is a task in the implementation cycle of distributed systems but not of centralized systems. An instantiation of the spiral for distributed development may contain entire new cycles.

A less remarkable but still useful insight is that recognizing the need to identify and resolve risks, and to do fairly extensive analysis for risk resolution, puts resources (time, people and money) behind what people either did anyway, but without adequate resources or didn't do and paid for with a failed development or a poor product. This is a pleasant side effect of recognizing risk as the proper driver for a software process. The main benefit is that developers are forced to pay attention to risk identification and resolution.

3.3.3. An Integrating Infrastructure – Information Repository

3.3.3.1. Goals

All software development produces and uses masses of information. Management and full exploitation of this information is a key to cost effectively developing software that achieves desired system goals. The major information management issues are:

- Consistency
- Ability of tools that produce and/or consume the same information to obtain that information without "manual" translation by people.

Solving the information management problem effectively integrates tools already in an environment and prepares the environment to absorb new tools.

Because distributed system development involves more experimentation than centralized development, it produces and consumes even more information than centralized development. One cannot expect to succeed at development of distributed system by experiment without automation for information management. Therefore, we have determined that a project database is the appropriate integration infrastructure. This repository can contain all relevant information about the system. Consistency is maintained because every stored and derivable item of information is written only once and because the semantics of that item is established by the repository for all tools that access it. Information is available to all tools that need it without human intervention because all tools consume input from and produce output to the same repository. New tools are easily absorbed into the environment; they use the repository either directly or with appropriate tools adapters.

The overall goal of tool integration is to support the combination of tools in an integrated software development environment, facilitating cooperation among tools and making it convenient for application developers to use. All integration solutions lower the effort people have to expend to apply tools by raising the level of automation that supports tool application. Support can take the form of a standard user interface, modeless switching among tools, automated data exchange, or a development process advisor that suggests what tools to apply and the order in which to apply them.

The two main goals for tool integration in this project are:

1. Support and encourage cooperation among independently developed tools and tool sets by automating (or partially automating) data exchange and sharing among tools. This goal seems most appropriate for this project because:

   - The DISE environment will acquire and support tools from many sources, militating against requiring all tools to work from a single internal representation.
   - The tools will perform distinct classes of functions (compiling, run-time monitoring, design and implementation time analyses) that need distinct kinds of data (not just forms of the program) again militating against a single program representation as the only source of data.
   - Eventually, DISE will include and support project management tools and tools that cannot foresee. To accommodate these kinds of tools that might use data in forms not previously anticipated, the framework/infrastructure has to be extensible.

2. Capture and retain all information needed to develop the application. Information capture and retention is essential for any environment. A development environment's utility is directly proportional to the number of activities it automates (or partially automates). Automation of a task can happen only if all the information needed to do the task except what the users supply interactively is in the system/computer. This extensible/open framework has to not only be able to accommodate new/foreign tools but can also acquire by extension all information that may be needed during system

development.

The mechanism that provides tool integration in the DISE environment must exhibit the following attributes:

- open : it should support all existing tools and should be able to absorb/accommodate new and foreign tools
- extensible : it should be able to provide novel services
- development process and language independent : because there is no widely accepted development process, design language or implementation language for distributed applications, and a single method and language cannot be prescribed for the use and development of all tools and systems, the integration infrastructure must not depend on characteristics of any process or language.
- useful : provide a certain degree of integration given the above constraints, provide a high level of service for new tool developers and guiding design principles for new tools, mechanisms or tool/data adapters that facilitate integration of new and existing tools.
- easy to use : be user friendly, provide a *comfortable* view of the tools and not require the user to learn/re-learn new and different commands/techniques to be able to work in/use the environment.
- cost effective : as inexpensive as possible for the services it provides - to develop, to use when initially integrating the existing tools, to use on an on-going basis and to integrate new/foreign tools.

3.3.3.2. Approaches to Tool Integration

Mechanisms for tool integration can be provided in several ways and at different levels. Several prototypical and theoretical mechanisms have been developed ranging from relatively simple and straightforward extensions to the operating system (UNIX pipes) to tightly coupled tools as in the Interlisp environment.

These approaches to tool integration can be classified based on the extent or amount of integration needed. The simplest of these are based on operating system technology where the data is stored directly in the file system, there is hardly any coordination of the tools used and almost no facilities for the retention and exchange of information among the tools. Examples of this may be UNIX pipes and input-output redirection facilities that allow related tools to interact in a very uncoordinated way.

At the other extreme, syntax-directed and structure oriented environments provide a much tighter coupling of tools around a single language and usually a single internal form of the program, and provide mechanisms and rules for the way the tools interact, the information they share, etc. However, these environments are language dependent, not very easy to extend and are characterized by the tools they support, which are mainly for the construction and manipulation (editing, compilation, execution) of programs. Moreover, they do not provide much support for programming-in-the-large. Editors, compilers, debuggers, etc., operate in total harmony and are integrated to the extent that tool boundaries almost disappear, and the user is sometimes not even aware that all these tools are participating in the software development process. The Interlisp environment, Smalltalk and Cedar are examples of this kind of integration [Penedo88a].

Tool integration/interaction may also be based on tool adapters that act as syntactic and semantic translators for the information shared/needed by two tools. Other mechanisms are based on an infrastructure or mechanism for (semi)automated data exchange/sharing among tools. They contain a database management system with the database serving as a (logically) centralized repository of information. Data are structured through the use of schemas, and the supported tools operate by transforming data, checking data in and out as needed.

Environments structured like these provide a rather loose level of tool integration. However, systems such as these are *open* - easily extensible - and support programming-in-the-large by providing mechanisms/facilities that support version control, configuration management, etc. Some of the other advantages of such an approach are: it simplifies tool integration as each tool builder has to think about just one interface - with the database; information is retained in the database; it is easier to maintain information about relations between the different program entities; consistency of the information can be more easily imposed/checked/enforced; inference information is available; information about the different stages of system development can be kept, and a software process can be applied to coordinate the actions of tools.

3.3.3.3. Approach Chosen

In the context of the integration goals and requirements enumerated above, these different tool integration mechanisms can be evaluated with respect to the need for extensibility, utilities provided and the level of integration desired. The approach we have chosen is to facilitate data exchange/sharing via a common project database, an approach that is between the two extremes of no integration/no constraints (e.g., UNIX pipes) and *fully* (highly) integrated constraining mechanisms (e.g., InterLisp).

This loosely integrated framework permits addition of new entities/data items in the database, and permits the integration of existing and new/foreign tools. The approach allows the integrating mechanism to be *open-ended* and capable of evolving over time as the technologies of environments and host systems evolve. The schema can be extended to contain new kinds of data as long as the data modeling language is sufficiently rich to model their semantics, its functionality can be enhanced by introducing new tools such as ad hoc query processors and new operations on the data. The approach also imposes no restrictions on tool writing methodologies, and interfaces to the database can be designed to be usable by any tool, regardless of methodological considerations [Obernd88a]. The project database is insensitive to the syntax of the data it contains and so the use of a database to do a task such as software development is orthogonal to the process used to do the development. This open-endedness is essential to permit/support cooperation among independently developed tools by automating or partially automating data exchange among tools. No language/system development method is imposed as it accommodates different methods and languages. These issues are especially important in the DISE environment, where tools will be independently developed possibly using different methods, different languages may be in use and system development methods cannot be imposed.

Most current systems do not place sufficient emphasis on the integration of the different phases in the development cycle. This conceptual database approach facilitates the retention of information - both technical and management - developed and used at different stages and encourages the disciplined and coordinated application of tools. The database is used as a repository of information used by the tools. The kinds of data items and their granularities will be largely decided by the kinds of tools the database will support, the kinds of information they require and their I/O requirements.

3.4. Candidate Development Tools

Previous sections discussed some of the unique characteristics of distributed systems and identified some of the particular risk areas which must be addressed in the development of distributed applications. The key message from those studies is that automated technical tools are needed to assist developers in dealing with the complexities introduced by the distributed nature of these applications.

There are two dimensions in the spectrum of technical tools, one covering the various stages of software development and the second representing distributed program risk areas. With respect to the first dimension, technical tools must be available at all stages of development, from specification through implementation and maintenance. Distributed debuggers exemplify a tool applied at the later stages of development, while a design assistant tool would be utilized in the earlier stages.

As for the second dimension, technical tools must also address the various risk areas inherent in distributed program development. A performance monitoring tool would address the performance risk area, while a tool for identifying concurrency anomalies addresses the correctness risk area. Identified below are the definitions of some candidate tools that are deemed to be necessary and important in DISE.

The investigations established that while a variety of implementation-time tools (for example, distributed debuggers) are being designed and implemented by the distributed systems research community, there seems to be a distinct lack of tools that can be profitably applied at *design time*.. Inability to verify or evaluate the design is a critical problem in the development of distributed applications. This is especially unfortunate because many of the problems related to the distributedness can be resolved, at least partially, during design.

Pre-implementation tools are deemed vital by the developers of distributed software, as they can facilitate the definition and analysis of an application's design, leading to improved implementations and overall lower development costs. We envision a subset of tools providing assistance from design synthesis through design analysis as representing a significant benefit to developers of distributed applications. The tools identified below include both design time and implementation time tools.

3.4.1. Design Assistant

Description

The design assistant tool is an interactive tool which assists the developers in synthesizing the designs of distributed applications. This tool provides templates to accommodate the design specification of system components such as objects. The tool is applied at design time prior to any coding. A graphical interface allows designers to visualize and experiment with the designs.

It is likely that the number of objects for a distributed application is large and there are many interactions and dependencies among them. Some cross references and type checking of the definitions of objects and their operations can be performed to maintain consistency. Depending on the types of studies desired, the design model should allow necessary information to be specified. Also, should any modification made to the object specifications, any inconsistency created would be flagged. This tool also allows the users to focus on a particular group of components in the system under development, say, in the examination of particular relationships between objects.

Input

Developers supply high level specification of objects including their operations, processing workloads, and interactions between objects.

Output

The tool generates graphical representation of the design (e.g., control flow graph constructed based on the design specification). Interactions between the objects can be depicted with directional arcs. Object interface specifications (without bodies) augmented with data (e.g., workloads, interactions, etc.) which may be used by other tools, are preserved for use by other tools.

Payoff for Developers

Correctness: The interactive, template-driven nature of the tool allow the developers to design in a systematic manner and therefore reduces potential for costly redesigns at a later stage. Coordination of information available at design time combined with the ability to focus on a particular subset of the related units facilitates the correct development of large distributed applications.

Performance: Indication of possible performance problems such as gross computational bottlenecks can be realized by examining the graphical display of the design in the context of workload and interaction patterns. This leads to higher performance of the implemented application.

3.4.2. Performance Analyzer

Description

The performance analysis assistant evaluates some performance aspects of the high level design of distributed applications. Aspects of performance can be addressed at design time are the granularity of decomposition and the overhead associated with the communication mechanism used. The performance analysis assistant allows the developers to study the followings and improve design iteratively:

- The effect of different granularity of decomposition on the system's performance based on the high level description of system components expressed as units of processing such as objects.
- How a system spends its time and the interaction patterns between components which might lead to processing bottlenecks.
- The effect of the communication mechanisms used.
- An extension would allow the cost analysis associated with a specified degree of fault-tolerance. For example, this would allow an evaluation of the cost of achieving a certain degree of availability of system objects.

The design can be analyzed either by mapping it to some computational model such as Petri nets or by simulation. Analysis of the profiles of the processing units can lead to determination of processing bottlenecks, and detection of certain concurrency anomalies. Global patterns of execution can also be simulated to obtain gross estimates of the system behavior.

Input

High level description of units of processing such as workloads, processing time requirements and communication costs between interacting units is used as input to the tool.

Output

Estimates of the system performance can be made by applying this tool. The process of measuring performance usually involves timing measurements which may lead to identification of bottlenecks by determining where the majority of elapsed is spent during certain operations. Analysis of the design to answer issues such as concurrency and reachability can be obtained if the underlying computational model has sufficient power.

Payoff for Developers

Performance: The granularity of decomposing a large complex system into processing units will have significant effect on the system performance. A poor decomposition (units too large, processes too small, etc.) may lead to inefficient patterns of execution and poor resource utilization. Furthermore, decomposition will affect the synchronization structure of the system, and a poorly synchronized system may exhibit increased response time. This addresses performance risk since this tool assists with the design of efficient applications. Early treatment of the performance issue is critical in order to avoid unacceptable performance and/or costly redesign at implementation time.

3.4.3. Execution Monitor

Description

This tool monitors a distributed program during execution. Program modules (units of distribution) are compiled with monitor code which generates messages to a centralized display console and/or files. The user is responsible for inserting monitor calls into the source code. It is a passive, invasive approach to debugging; passive in the sense that the user cannot interactively control program execution through the display, and invasive since monitor code is incorporated into monitored modules. It is a validation/verification tool, not a development tool in the classic sense.

Input

The monitor requires instrumented source files as input. User commands are received interactively.

Output

Simple textual dumps, graphical displays, and statistical analyses could all be supported output formats. The tool eliminates the need for multiple terminal sessions for monitoring the distributed modules, and provides coordination and processing from a single display console.

Payoff for Developers

Correctness: The tool addresses the risk area of not being able to determine what the program is doing and how it is doing it.

Performance: The tool assists the user in identifying performance bottlenecks and resource utilization.

3.4.4. Distributed Debugger

Description

Debuggers that are available in centralized systems are not adequate for debugging distributed software. A distributed debugger must take into account the interaction between processes. Some of the approaches to the debugging of distributed software that have been investigated are:

- Behavioral abstraction and EDL (Event Definition Language)
- Dynamic analysis
- Post-execution examination
- Distributed monitoring
- Interactive break-examine

However, [Garcia84a] acknowledged that a fully implemented and complete distributed debugging system does not yet exist. If a general-purpose distributed debugger is ever successfully developed, it will probably comprise the following capabilities:

Ability to represent program state

The ability to represent program state is a fundamental capability of debugging. A conceptual model of expected system behavior is needed to make a meaningful comparison between actual system activity and expected system behavior. Abstraction of fundamental and significant system behavior can provide an easier means in describing the system behavior and thus aid in detecting the differences in implementation and the expected behavior.

Ability to examine program state

Since the observed program state in a distributed environment is often hard to reproduce, a facility to record the events when a bug is observed for examination later is needed. Tracing can be implemented such that the set of commonly occurring events in a distributed program is recorded automatically. The capability to incorporate trace entries inserted by the programmers can be of great value. However, tracing and replaying are often very time consuming. A mechanism to simplify the task of examining trace records must be considered. [Garcia84a] proposed simplified distributed database techniques to examine the trace entries.

Ability to monitor program state

In the distributed program, it is often impossible to specify exactly the state of the desired breakpoint or of the possible failures. Assertion monitoring is a useful technique in catching some time dependent errors by allowing automatic monitoring of states and message sequences, as opposed to the manual insertion of breakpoints. It also relieves the programmer of the burden of examining large traces. However, it is difficult to define sufficient assertions that characterize all of the possible errors due to the complexity of distributed programs and the available means of formalism. [Harter85a] argued that assertion monitoring is too powerful to ignore in a distributed environment.

Input

The input to the debugger would include object code compiled with a speciai option (i.e., -g option) to produce additional symbol table and other context information for the determination of the variable values.

Output

A graphical display of process interactions is suggested for assisting the user in interpreting the data collected. Graphic representation of the interactions between the processes or textual display of the information collected can help in visualizing the system control flow. Like centralized debuggers, this tool is highly interactive.

Payoff for Developers

Correctness: After a program is written, various test cases are run against a set of stated or implicit expectations. Given the evidence of bugs, this tool is used to diagnose the cause of the problem so the errors can be removed from the software. It clearly addresses the performance risk.

3.4.5. Allocation tool

Description

The allocation tool determines an assignment of distribution units to processing nodes which meets objectives. While different assignments of units to nodes va with respect to performance and/or fault tolerance, they are all functionally equivalent -- that is, the program implementations accomplish the same thing but some may exhibit better performance and/or a higher degree of fault-tolerance than others.

In general, there are a combinatorially large number of ways to map units onto processors. Furthermore, the general problem has been shown to be inherently complex in that no practical algorithms exist for finding the optimal solution. This implies two things for an allocation tool developer: approximation and/or heuristic algorithms will have to be employed for finding a decent, suboptimal solution, and an evaluation model must be developed for quickly estimating the quality of a given assignment of units to nodes.

The allocation tool is applied after coding and before execution. In a heterogeneous system, allocation would occur prior to compilation since the type of object code generated and system calls used would depend on the target machine for each unit. In a homogeneous system, allocation could occur after compilation.

Input

Workload estimates for all units must be provided to the tool. These quantities include both processing and communication loads. The user may provide these parameters directly, or a front end code analyzer could generate them. Also, models or attributes of processors and the communication subsystem must be provided.

Output

The allocation tool generates a mapping from distribution units to processing nodes. This information can be utilized by a program invocation tool which distributes and initiates the appropriate processes (objects and clients) on the designated nodes.

Payoff for Developers

Performance: The allocation tool directly addresses performance risk area as described in the risk area study. There, allocation was described as a key area of concern with respect to achieving performance objectives. An allocation tool can generate program implementations which make effective use of processing resources, realize parallelism between units, and have units assigned to the nodes for which they are best suited (e.g., with respect to memo, floating point capability).

The allocation tool also mitigates the communication cost aspect of performance by incorporating communication costs into its internal models of program cost. Allocation schemes are generated that minimize the negative effects of high communication costs.

Failures: An allocation tool may also address the failure risk area, in particular through the determination of a replication strategy for each unit. Data availability (hence system reliability) can be enhanced through replication. Furthermore, performance gains can be realized through locality of reference -- if the unit (e.g., object) is replicated on a "closer" site, it will cost less to access it. At the same time, however, performance can be degraded due to the cost of maintaining data consistency and concurrency controls across multiple copies. These tradeoffs (reliability and efficiency versus maintenance) can be incorporated into the cost models used within the allocation tool, thereby allowing the degree of replication for each unit to be determined as well as the placement of replicates.

3.5. References

Babaog87a.

O. Babaoglu, "On the Reliability of Consensus-Based Fault-Tolerant Distributed Computing Systems," *ACM Transactions on Computer Systems* **5** pp. 394-416 (November 1987).

Bates82a.

P. Bates and J. C. Wileden, "EDL : A System for Distributed System Debugging Tools," *Proceedings of the Hawai International Conference on System Science,* (Jan 1982).

Bhatt88a.

D. Bhatt and et. al., "The Instrumentation of an Experimental Distributed Computer Testbed," *IEE Technical Committee on Distributed Processing Newsletter* **10** pp. 11-19 (March, 1988).

Boehm83a.

Barry W. Boehm, "Seven Basic Principles of Software Engineering," *The Journal of Systems and Software* **3**(1) pp. 3-24 (March 1983).

Boehm86a.

Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," *Software Engineering Notes* **11**(4) pp. 14-24 (August 1986). (Special issue for the International Workshop on Software Process and Software Environments.)

Bottin86a.

R. Botting, "Novel Security Techniques for on-line systems," *Communications of the ACM* **29** pp. 416-419 (May 1986).

Chandy85a.

K. Mani Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of

Distributed Systems,'' *ACM Transactions on Computer Systems* **3** pp. 63-75 (February 1985).

Chaum85a.
D. Chaum, ''Security Without Identification: Transaction Systems to Make Big Brother Obsolete,'' *Communications of the ACM* **28** pp. 1030-1044 (October 1985).

Chikof88a.
Elliot J. Chikofsky, ''Software Technology People Can Really Use,'' *IEEE Software* **5**(2) pp. 8-10 (March 1988). (Guest editor's introduction to the special issue on CASE)

Dasgup88a.
P. Dasgupta and et. al., ''The Clouds Distributed Operating System,'' *IEEE Proc. 8th International Conference on Distributed Computing Systems*, pp. 2-9 (June 1988).

Feldma79a.
J. A. Feldman, ''High Level Programming for Distributed Computing,'' *Communications of the ACM* **22** pp. 353-367 (June 1979).

Garcia84a.
H. Garcia-Molina and F. Germano, Jr., ''Debugging A Distributed Computing Systems,'' *IEEE Transactions on Software Engineering* **Vol. SE-10, No 2**(March 1984).

Garey79a.
M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co. (1979).

Goyal88a.
A. Goyal and A. N. Tantawi, ''A Measure of Guaranteed Availability and its Numerical Evaluation,'' *IEEE Transactions on Computers* **37** pp. 25-32 IEEE, (January 1988).

Harter85a.
P. Harter, D. Heimbigner , and R. King, ''IDD: An Interactive Distributed Debugger,'' *Proceedings of the 5th International Conference on Distributed Computing Systems*, (May 1985).

Kohler81a.
W.H. Kohler, ''A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems,'' *Computing Surveys* **13** pp. 149-177 (June 1981).

Micros86a.
Sun Microsystems, Inc., *Networking on the Sun Workstation*, Sun Microsystems, Inc., Mountain View, CA. (February 1986).

Needha78a.
R. M. Needham and M.D. Schroeder, ''Using Encryption for Authentication in Large Networks of Computers,'' *Communications of the ACM* **21** pp. 993-998 (December 1978).

Newton87a.
Anthony M. Newton and Richard H. Sweed, ''Distributed Systems Evaluation Environment: An Introductory Report,'' *RADC In-House Report* **RADC-TM-87-5**(April 1987).

Oberndorf88a.
Patricia A. Oberndorf, ''The Common Ada Programming Support Environment (ASPE) Interface Set (CAIS),'' *IEEE Transactions on Software Engineering* **14** pp. 742-748 (June 1988).

Patnai86a.
L.M. Patnaik and K.V. Iyer, ''Load-Leveling in Fault-Tolerant Distributed Computing Systems,'' *IEEE Transactions on Software Engineering* **SE-12** pp. 554-560 (April 1986).

Penedo88a.
Maria H. Penedo and William E. Riddle, "Guest Editors' Introduction: Software Engineering Environment Architectures," *IEEE Trans. on Software Engineering* **14**(6) pp. 689-696 (June 1988).

Pressm83a.
Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill Book Company, New York (1983).

Ramesh87a.
S. Ramesh and S. Mehndiratta, "A Methodology for Developing Distributed Programs," *IEEE Trans. Software Engineering*, pp. 967-976 (August 1987).

Randel72a.
B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering* **SE-1** pp. 220-232 (June 1972).

Randel78a.
B. Randell, P.A. Lee, and P.C. Treeleaven, "Reliability Issues in Computing System Design," *Computing Surveys* **10** pp. 123-165 (June 1978).

Schant86a.
R. Schantz and et. al., "The Architecture of the Cronus Distributed Operating System," *IEEE Proc. 6th International Conference on Distributed Computing Systems*, (May 1986).

Shriva86a.
S.K. Shrivastava, "Structuring Distributed Systems for Recoverability and Crash Resistance," *IEEE Transactions on Software Engineering* **SE-7** pp. 436-447 (July 1986).

Silver88a.
J. Silverman, *HOPS: Honeywell Object Programming System.* January 1988.

Stoneb79a.
M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies in Distributed INGRES," *IEEE Transactions on Software Engineering* **SE-5** pp. 188-194 (May 1979).

Tanenb81a.
A. Tanenbaum, *Computer Networks*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1981).

Traige82a.
I.L Traiger, J. Gray, C. A. Galtieri, and B.G. Lindsay, "Transactions and Consistency in Distributed Database Systems," *ACM Transactions on Database Systems* **7** pp. 323-342 (September 1982).

Weber83a.
J. Weber, "Interactive Debugging of Concurrent Programs," *ACM Proceedings*, (1983).

SECTION 4

Tool Integration Platform

This section discusses the purpose and architecture of the DISE Tool Integration Platform.

4.1. Introduction

The purpose of the DISE Tool Integration Platform (IP) is to:

- support and encourage cooperation among (existing and new) independently developed programming tools by automating (or partially automating) data exchange and sharing among the tools, and
- capture and retain all information required to develop the application/system (information generated during system development).

By raising the level of automation supporting tool application, tool integration lowers the effort developers have to expend to use tools. The IP represents a structured means for introducing new tools into the environment and supporting and encouraging cooperation among tools Information capture and retention is essential for any environment: A development environment's utility is directly proportional to the number of activities it automates (or partially automates). Automation of a task can happen only if all the information needed to do the task (except what users supply interactively) is online. The integration platform must contain (or be able to contain through simple extensions) all the information DISE developers need today and will need in the future.

Implementing the IP involves two primary tasks:

- developing an information model that reflects the entities of interest to development tools; and
- building software that provides tools with runtime access to project data.

Both aspects of IP implementation are discussed below.

4.2. Information Model

An information model is a formal (or semi-formal) specification of the data comprising some particular domain. It is independent of any particular database schema. The information model underlying the IP is geared towards the development of distributed software in the DISE environment. It models general distributed system concepts such as units of distribution and processors, and DISE-specific concepts such as the contents of *.typedef* and *.mgr* files that the Cronus tools *definetype* and *genmgr* accept as input.

Figure 3 shows a small excerpt from the information model. The modeling constructs used to define the IP's information model are based on an entity relationship model fitted with some object oriented concepts. The model contains entities (ovals), and entities have both attributes and relationships (lines between ovals). Entities are structured into inheritance hierarchies (bold arrows between ovals) where the children inherit all of the parent's attributes. The expressiveness of this model greatly facilitates the complex process of developing an information model for the diverse set of types that exist in this environment.

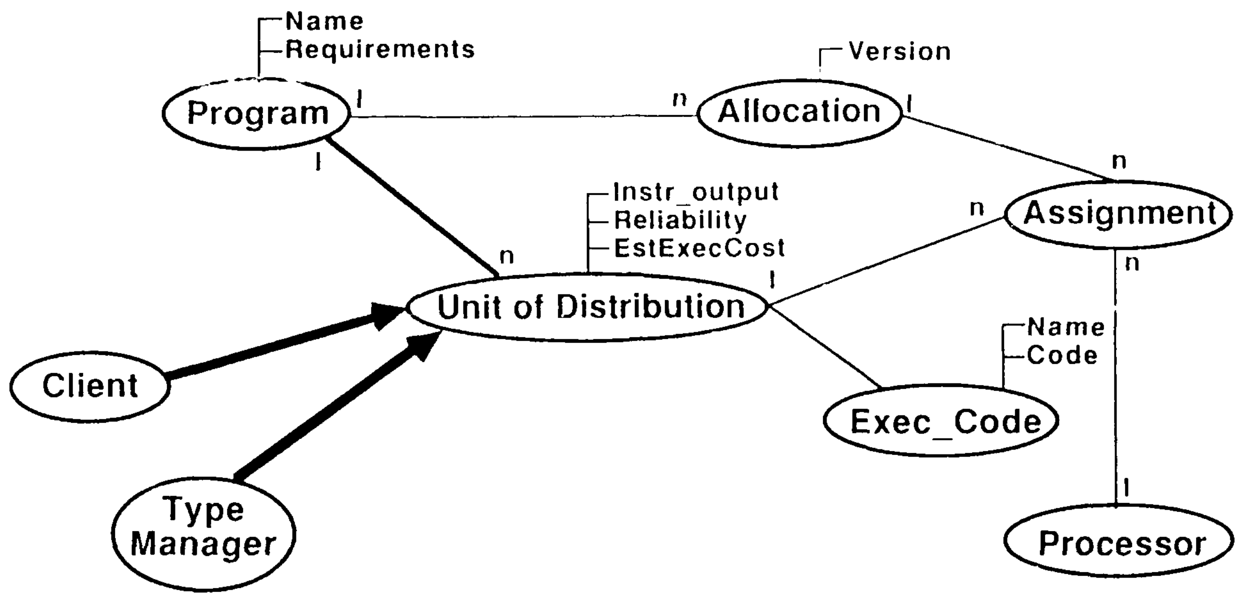


Figure 3. Information Model Excerpt

To get a flavor of the model's elements and its use in the IP, consider the sample information model shown in Figure 3. A **Program** has Name and Requirements attributes and consists of (is related to) a set of **Units of Distribution** (note the 1:n mapping indicated on the relationship arc). A **Program** object also consists of a set of **Allocations.** An **Allocation** represents the mapping of program components to processors in the execution environment, and consists of a set of **Assignments**. An **Assignment** object captures the relationship between a particular **Unit of Distribution** and the **Processor** to which it has been assigned for execution. Finally, note that **Client** (a Cronus application) and **Type Manager** are sub-types of **Unit of Distribution**, from which they inherit all the relationships and attributes associated with **Units of Distribution**.

We have modeled from both the Cronus point of view and the general software engineering point of view to support software engineering within the Cronus world. Cronus development tools work on files: definetype uses .typedef; genmgr uses .mgr and produces nine output files that are eventually compiled. Those files have to be used *as files* to produce a program. Software developers have come to think of the product they are working with as residing in particular files. In fact, the artifacts they work with have to reside in files only at particular times in their lives (e.g, for compilation, for use by genmgr and so forth). On the other hand, the software development style we are trying to promote is based on a view of working on information structured the way developers logically model a program (as types, procedures, and so forth). For example, we want users to examine a type, not the file containing a type, to learn something about the type.

The IP bridges both worlds. It stores things as objects in the software development world, and, with the right tools, can generate them in object form for examination and editing. But it can also generate the files that Cronus tools need, just as it can generate data that integrated tools such as the Allocator need. The IP stores not only all the data that comprise the software product, and information about the product such as the performance of a type manager, but also mappings of the objects into files, to

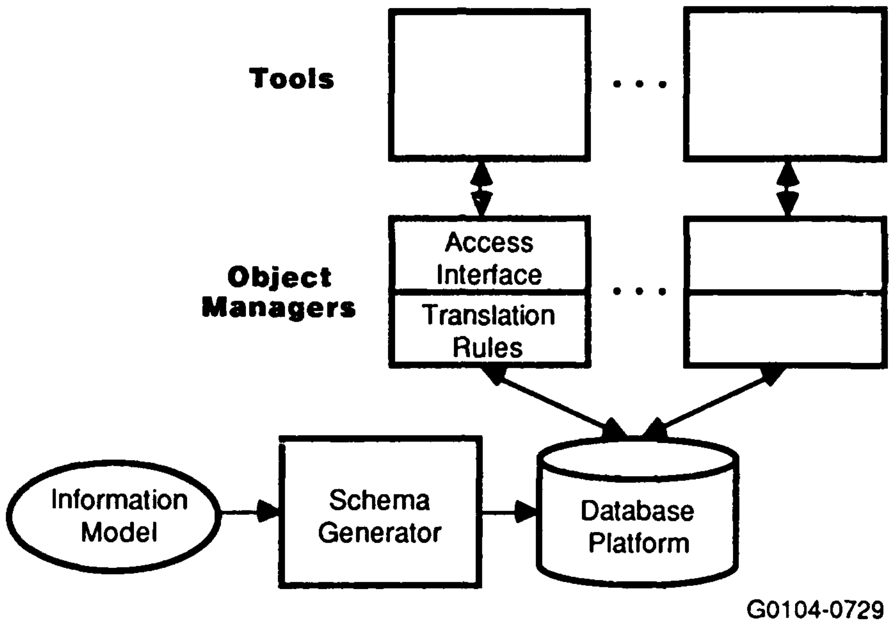


Figure 4. Integration Platform Architecture

effect the file view. The mappings are implemented by methods attached to the objects Cronus-file and File-element. The methods are structured according to a syntactic (BNF) description of Cronus files.

4.3. Integration Platform Architecture

The IP architecture is depicted in Figure 4. It consists of two basic components: a tool object manager through which tools access data in the IP, and an underlying database platform.

An object manager exists for each type of tool in the environment. It provides a set of data access and storage operations that are tailored to the needs of that tool. For example, an operation might be defined that retrieves a particular subset of processing environment attributes (such as computing speeds of processors). Or an operation might return the contents of a particular source code file, or might store application performance data generated by the tool into the IP. The operations are invoked by the tool through the use of the remote procedure call (RPC) mechanism provided in Cronus.

These operations are implemented in terms of a set of translation rules. These translation rules define a mapping between the external (tool) and internal (IP) views of the data. The rules translate external, tool-specific data references into the appropriate internal database references. These rules are written using the database's programmatic interface. Object manager operations invoke these translation rules to effect changes on the database and to retrieve data from the database.

The underlying database platform, which could in principle be any database software package (e.g., relational or object-oriented), provides basic database functionality such as permanence and transaction processing. It includes a programmatic interface and schema definition facilities. The IP's information model, described above, is translated into a schema using the database's schema definition facilities, which include a data definition language and a schema compiler.

Some of the advantages of this architecture are:

- All issues relating to distribution (message routing, location transparency, naming, etc.) are handled by the Cronus distributed operating system. The database can, therefore, support tools that run at remote sites as well as tools that are themselves distributed.
- By encapsulating all database implementation dependencies, a consistent external interface can be provided to the tools. The different tools in the supported toolset will interact with the database through the tool object managers only, and therefore are not dependent on the nature of the underlying database. That is, tools see the IP just like any other distributed system service. As all Cronus applications interact with Cronus managers/objects, this architecture allows the different tools running as Cronus applications to access data in the database via a similar programming interface.
- Changes to the information model do not impact the tools or their object manager interfaces. Only the translation rules, possibly, would be impacted.

4.4. Integration Platform Implementation

The implementation of the Integration Platform, depicted schematically in Figure 5, varies slightly from the architecture just described due to a number of implementation constraints. The differences between the two revolve around the nature of the particular object oriented database employed and the manner in which translation rules interface to it.

The object-oriented commercial database ONTOS[1] has been incorporated into the IP as the database

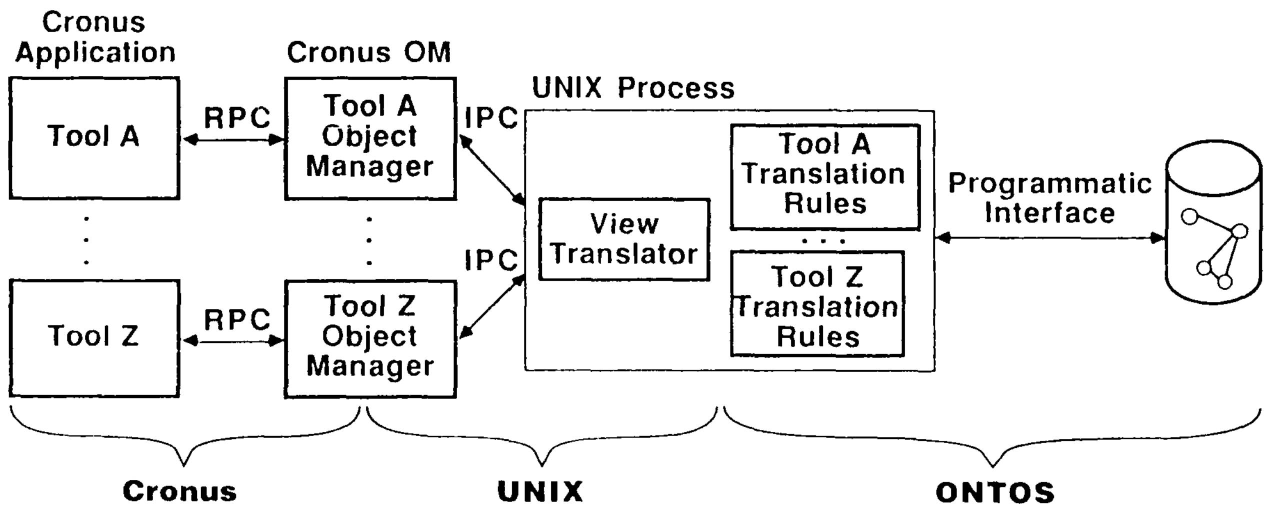


Figure 5. Integration Platform Implementation Schematic

[1]ONTOS is a trademark of Ontologic, Incorporated.

substrate. ONTOS provides transaction processing and permanence for objects defined using the language C++ and its class definition facilities. The schema derived from the information model is expressed as a collection of C++ classes (in the C++ language, a class is the definition of an object type). Tool translation rules are manifested as C++ applications that create and access database objects using the programmatic interface provided by ONTOS.

Use of an object-oriented database has greatly facilitated the process of translating the information model into a database schema. Entities defined in the information model are typically mapped directly

```
class UOD : public dosObject {
  // This class defines the Unit of Distribution entity.

  private:
    /* Attributes */
    char   *Instr_output;
    float  Reliability;
    float  EstExecCost;

    /* Relations */
    Program      *program;
    Exec_Code    *exec_code;

  public:
    /* Constructors */
    UOD(APL *theAPL);  // for use by ONTOS
    UOD(char *Name=0, Program *prog=0); // called to make an instance

    /* Accessors */
    Program *program();
    Exec_Code *exec_code();
    char *instr_output() { return  Instr_output; };
    float reliability() { return Reliability; };
    float estimated_cost() { return EstExecCost; };

    /* Modifiers */
    char *instr_output( char *instr);
    float reliability (float rel) { return (Reliability = rel); };
    Program *program(Program *prog);
    Exec_Code *exec_code(Exec_Code *code);
    float estimated_cost(float c) { return (EstExecCost = c); };
};
```

Figure 6. Sample Schema Class Definition

into an object type declaration in the database. Figure 6 shows, for example, the ONTOS class definition for the **Unit of Distribution** (UOD) entity discussed earlier. Relationships between entities, both one-to-one and one-to-many, are readily mapped into the corresponding relations between objects.

Use of an object-oriented database substrate has also simplified the task of developing translation rules since direct access to object instances is supported and database traversals (a common operation in translation rules) are readily implemented. An illustration of this can be seen in Figure 7, which shows a translation rule written using ONTOS' programmatic interface. Given the name of a **Program** object, the translation rule retrieves some information about each of the **Units of Distribution** associated with that program.

Turning back to Figure 5, the implementation schematic differs structurally from the architecture in that the tool translation rules are grouped together into a single module called the View Translator. The View Translator processes all tool translation rules, essentially multiplexing requests from tool object managers. It is a single ONTOS application to which all tool object managers interface when they

```
int get_program_data(char *program_name, char *buf)
// This rule interates through the set of UODs associated with
// the Program object named "program_name" and records each
// UOD's name and estimated execution cost.
{
        UOD    *uod;   // reference to a Unit of Distribution object
        char   nextentry[25];

        Program *pgm = (Program *) OC_lookup(program_name);
        if (!pgm)
                return NO_SUCH_APPL;

        Set *uod_set = pgm->uods();
        if (!uod_set)
                return OK; // no UODs belong to this program

        // SetIterator is an ONTOS class used to iterate through sets
        SetIterator *next_uod = new SetIterator(uod_set);
        while (next_uod->moreData()) {
                uod = (*next_uod) (); // returns the next UOD object in the set
                sprintf(nextentry,"%s  %f ",
                        ((Object *)uod)->Name(),uod->estimated_cost());
                strcat(buf, nextentry);
        }
        return OK;
}
```

Figure 7. Sample Translation Rule

need access to the database. This simple interface is readily available to tool developers as a collection of built-in service modules.

The View Translator is a system entity that begins by opening a session with the database, then waits for requests from tool object managers and dispatches the appropriate translation rules in response to those requests. A translation rule is typically a single atomic transaction, but can be a sequence of individual transactions; a single transaction, however, cannot cross translation rule boundaries.

The primary advantages of this implementation are:

- The interface between the tool object managers and the View Translator defines the boundary between the database-specific IP components and database-independent components. Tool object managers have no database-specific code in them, so that they will not be impacted by changes in the schema or modification to translation rules; in fact, the database itself could be replaced by a new product without requiring any changes to the object managers.
- Related to the previous point, this approach avoids the problems associated with linking ONTOS modules with Cronus modules.
- Individual translation rules can be shared between different tools.
- Accesses to the IP do not involve process creation. The View Translator and object managers are processes that span tool IP request invocations, therefore no performance hit is absorbed due to process creation during tool IP requests.
- Accesses to the IP do not require database *Open* calls, which are very costly operations (typically 4 to 8 seconds in duration). The View Translator opens the database once and keeps it open across multiple calls from the same or different tools.[2]

The last two points are key with respect to enhancing the IP's performance. While extensive performance studies have not been performed, large amounts of data can be stored into or retrieved from the IP relatively quickly, typically 1 to 5 seconds. Considering the IP's performance from a user interface perspective -- in conjunction with the facts that disk access is involved, database functionality is being provided, and IP requests are made relatively infrequently during any single instantiation of a tool (typically at initiation and termination) -- this level of performance is acceptable.

Despite these advantages, there are nevertheless a couple of disadvantages associated with this implementation:

- The View Translator increases in size with the integration of each new tool.
- The View Translator does not support concurrent tool requests. While it multiplexes requests from tool object managers, these requests are handled sequentially. Said another way, translation rules do not execute concurrently.

With respect to the first point, there is no barrier to the option of breaking the View Translator up into separate agents, each of which handles requests from a particular subset of tool object managers. The separate View Translators could thereby be kept at an acceptable size.

As for the second point, while concurrency of database access is desirable, it does not imply that every single translation rule should execute concurrently. That is, performance improvement eventually reaches a plateau as the number of concurrently active transactions increases. Therefore, the best option regarding concurrency would be to have the View Translator manage a set of identical agents (separate processes), which handle translation rule invocations. As the View Translator receives requests from tool object managers, it would dispatch one of the agent processes to carry out the

[2]Note that despite the presence of a single database open operation, each tool request still represents the unit of transaction atomicity.

requested action. In this way, multiple tool requests (with the maximum equal to the number of agent processes) could execute concurrently.

4.5. Integrating New Tools

The IP's architecture is designed to facilitate the process of integrating a tool. This has been achieved by decomposing database access across separate, well-defined software modules. The tool object managers provide a database-independent set of services that collectively define a tool-specific view of the entire schema. The View Translator is a database-specific set of routines, invoked by tool object managers, built as simple, autonomous database applications. Finally, the schema itself has been designed with tool developers in mind:

- the information model is expressed using an intuitively appealing data model;
- each entity in the model represents a meaningful concept to the tool developer at an appropriate level of granularity; and
- the schema has been implemented using a common object-oriented language (C++).

There are two implications associated with the latter point. First, use of a standard programming language means that developers are not forced to learn a database-specific data manipulation language or interface. Second, the expressiveness of the object model allows for a very clear mapping between elements of the information model (entities and relationships at the conceptual level) and the schema (objects and references at the implementation level) as illustrated by the earlier examples. The information model is what tool developers read and understand, while the schema is what they use when implementing translation rules. This clear mapping makes translation rule development easier.

Integration of a new tool essentially involves four steps:

1) Determine the tool's data requirements with respect to the IP;
2) Develop a tool-specific object manager by first defining the IP operations needed by the tool, inserting calls on these operations into the tool, and finally implementing them in terms of translation rule invocations;
3) Implement the translation rules using the database's programmatic interface;
4) Start the tool's object manager.

After these steps have been performed, the tool will have runtime access to the IP for all its data requirements. As other tools store data into the IP, that data will become accessible to the new tool. Conversely, as the new tool consumes and stores new data into the IP, other integrated tools will have access to that new data.

4.6. Summary

The DISE Tool Integration Platform supports tool integration by providing a canonical data repository through which tools store and share information. It consists of an information model, a database substrate and tool object managers. New tools can be readily integrated and the information model can be easily extended to handle new data types. The IP appears in the system to be just like any other distributed operating system service, in keeping with the current development model.

The IP enables tools to cooperate in useful ways without the tool developer having to know the details (I/O formats, semantics, etc.) of every other tool. Tool interoperability arises through the definition of a canonical information model to which all tools appeal. Integration of new tools is thereby facilitated; every tool developer has full access to the single, canonical information model.

The IP is but one component of a larger integrating framework. However, it is a critical component and serves as a basis upon which additional framework elements (such as automated development methods) can be built.

The IP has been implemented so as to facilitate technology transfer. It uses a readily available commercial object oriented database accessed at runtime through Cronus object managers.

SECTION 5

Development Tools

This section briefly motivates the need for technical tools and the purpose their development serves under this contract. We then describe the two tools that have been implemented.

5.1. Motivation

Developing distributed programs is more difficult than developing centralized programs due to the problems associated with distributed processing. Problems that must be addressed include:

- Communicating between program components;
- Accommodating platform heterogeneity;
- Monitoring distributed execution;
- Managing concurrent/parallel execution;
- Verifying/debugging;
- Utilizing distributed resources;
- Minimizing communication overhead;
- Implementing fault tolerance;
- Decomposing the program into units of distribution.

This list represents just a sampling of the complex issues that must be addressed in the development of distributed applications. Successful resolution of these issues demands support in the form of software development tools, for example, distributed debuggers, instrumentation tools and performance analysis tools. Software tools can help to mitigate the problems brought about by distribution.

5.2. Objectives

Tools are being developed on this contract to achieve two primary objectives:

- To demonstrate technology that should be made available to distributed application developers; and
- To demonstrate the benefits and use of the Tool Integration Platform (IP).

In addition, the highest payoff tools are those that support the design stage of development, where distributed application developers are facing a particular shortage of automated support.

5.3. Results

Jointly with RADC we selected two tools suitable for achieving these goals. They are the Allocator and the Reliability Analyzer. The Allocator determines efficient allocations of program modules to processing nodes, and the Reliability Analyzer computes limiting availabilities for application components executing in faulty environments.

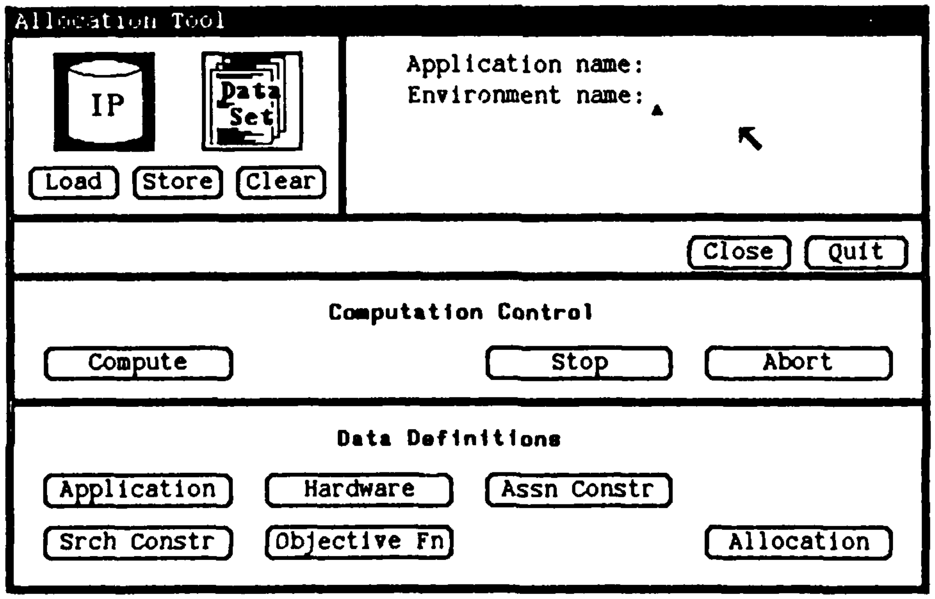
In addressing problems unique to distributed applications, these tools are representative of the type of technology that supports the development of distributed applications in DISE. They support design time analysis of an application's performance and reliability characteristics, which helps the developer deal with problems early in development when modifications are relatively inexpensive. Furthermore, since the two tools have certain I/O data in common, they provide the basis for a convenient demonstration of tool integration via Tool Integration Platform. As fully integrated tools, their implementation serves as rigorous examples of the process of integrating tools.

Both tools offer a convenient, window-based, graphical user interface. The user interface for the Allocator is shown in Figure 8. The user interface allows the developer to define tool inputs, dispatch the computation and analyze outputs. Experimentation is facilitated by the structure of this interface. It also provides a specific interface through which the developer controls access to the IP.

By using the IP as the repository for tool I/O data, the developer deals not in terms of file and directory names but rather deals directly with the development data of interest through direct naming, e.g., using application names, object names, environment names, processor names and so on. Attributes of named objects are then readily available as needed by the tool. The IP maintains the integrity and structure of the data it stores so that the developer can concentrate at the conceptual level. It is therefore easier to use integrated tools and and to leverage the capabilities of multiple tools in a cohesive manner.

Below we discuss the motivation, functionality and implementation of these two tools.

5.3.1. Allocator Tool



Application name:
Environment name:

Figure 8. Allocator User Interface

5.3.1.1. Motivation

Distributed programs consist of a collection of units of distribution (UODs) that interact through the use of interprocess communication. Under an object-oriented model of distributed computing, such as that used in Cronus, a UOD is either an object or a client and may be assigned (statically) to execute on any processor in the processing environment. This environment consists of computing platforms interconnected by a communications medium and is consistent with DISE's model of distributed processing.

Distributed application developers must, at some point during development, address the question of how to assign the program's units of distribution (objects and clients) to the processing nodes. While the manner in which they are assigned will not affect the application's functionality, it will have major implications for objectives related to performance, fault tolerance, resource utilization and security. Assignments will vary with respect to how the program performs during execution since resource utilization profiles, communication loads, and parallelism are all influenced by the pattern of allocation. The allocation problem is an important concern in DISE where performance and fault-tolerance are critical objectives for distributed applications.

Unfortunately for developers, the allocation problem exhibits combinatorial complexity; finding the optimal assignment of application components to processing nodes is an inherently difficult problem. Subtle tradeoffs exist between resource utilization, parallelism and communication costs. Nevertheless it remains an important issue in the development of distributed applications in this environment. It is critical, therefore, that distributed application developers are provided automated support for analyzing and determining an appropriate allocation for the program under development.

5.3.1.2. Tool Features

The Allocator supports distributed application developers by determining an efficient assignment of application components (objects and clients) to processing nodes. The tool utilizes information about the distributed application, its components and their interaction patterns, as well as characterizations of the processing environment. It balances processing and communication costs in finding an appropriate processor assignment for each unit of distribution in the program.

The Allocator has two main input categories: the distributed application and the hardware environment. Information needed about the application includes the number of components (units of distribution (UODs)), their names, their patterns of communication, and their expected execution cost. For the hardware environment, the number and relative speed of each processor is needed as well as the network topology and its performance parameters. Given this information, the tool computes a processor assignment for each UOD (collectively, an allocation for the application) such that the resulting software/hardware configuration will exhibit efficient run time performance.

Several additional input items for the tool may be defined: Search constraints may be specified by the user to limit the extent or scope of the search for an efficient allocation. Constraints may also be placed on the assignments themselves. For example, the user may specify that a particular UOD be assigned to a particular processor. Finally, the user can select either of two objective functions by which relative performance is to be gauged: expected response time or expected total execution time. Response time is the expected wall-clock time required for the program to execute; total time represents the cumulative execution cost for the program across all processors (i.e., resource utilization).

In using the tool, the user is free to manipulate any of the input parameters to investigate its effect on program performance. For example, the user may add some additional objects to the application, or modify the nature of interactions between two objects, or modify the number of processors in the processing environment. A new allocation can then be generated so that the effects of parameter manipulations can be evaluated. Experimentation is an important capability for developers as they deal with the complexity inherent in the development of distributed applications. This tool supports experimentation by offering generality and flexibility with respect to the types of inputs accepted, and by providing a convenient interface through which they can be manipulated.

The Allocator can also be applied across development stages, from design to implementation. Early in development the tool can be used to evaluate alternative high-level program decompositions with respect to their resource utilization. At this stage, the developer may only have some initial ideas about the decomposition into objects and clients along with some rough estimates of their expected resource utilization. The user can provide this information to the tool and get a rough idea of how well the program might execute in a given processing environment.

The tool can continue to be used effectively as the program's development proceeds. Communication patterns between UODs and execution cost estimates for each UOD will become more refined as development progresses. After coding, instrumentation tools can be used to generate actual performance data instead of estimates. These data can be fed into the Allocator and used in generating an efficient allocation based on the more accurate input data.

The Allocator is an integrated DISE tool. Under user guidance, it can access the DISE Tool Integration Platform (IP) for its input and output data requirements. As noted above, this simplifies the development process by allowing the developer to concentrate on the objects of interest rather than having to be concerned with input data formats and maintaining consistency across a collection of input files.

5.3.1.3. Implementation Notes

The allocation problem is computationally complex; deterministic algorithms for finding the optimal allocation for an arbitrary application and target environment might take many centuries to execute -- even for modest sized programs and environments. In computing efficient allocations, the Allocator therefore employs a combination of heuristic algorithms, including a hill-climbing algorithm, geared towards finding "decent" solutions in reasonable time.

The Allocator is a Cronus application written in C. It accesses the IP through Allocator's IP Cronus type manager. The tool's user interface is developed using SunView[3] and executes on Sun workstations.

5.3.2. Reliability Analyzer

5.3.2.1. Motivation

The DISE environment supports the development of C^3 applications for execution in distributed processing environments. One of the most important characteristics of an effective C^3 application is *survivability.* Survivability is the ability to meet mission requirements in the event of hardware failures and can be measured in terms of reliability and availability. Successful development of *survivable* applications requires the ability to evaluate their reliability and availability characteristics.

The reliability of an application can be enhanced by various software mechanisms such as atomic transactions, concurrency controls and replication. The developer must balance the benefit of these mechanisms against their inherent costs. Moreover, development decisions concerning reliability made at a later stage of development, such as implementation, that lead to unacceptable levels of reliability can result in very costly redesigns. Therefore, it is important that distributed application developers have

[3]SunView is a trademark of Sun Microsystems, Incorporated.

the capability for design-time analysis of reliability characteristics.

5.3.2.2. Tool Features

The Reliability Analysis Tool allows the developer to analyze the implications of processing environment (hardware) reliabilities on distributed software applications. It supports the developer in experimenting with alternative design decisions related to the application's decomposition, component functionality, processing environment and allocation of software components to computing platforms, each of which influences application reliability characteristics. The Reliability Analyzer provides the ability to evaluate reliability characteristics of distributed application designs, allowing developers to build applications that are partitioned into an appropriate set of communicating objects and meet reliability objectives.

The Reliability Analysis Tool accepts input data through a convenient user interface and generates the expected reliability of the software components measured in terms of limiting availability (A) defined as:

$$A = \frac{MTTF}{(MTTF + MTTR)}$$

where MTTF is the mean-time-to-failure and MTTR is the mean-time-to-repair of the component. The tool computes this quantity for any application component selected by the tool user. The computation of the limiting availability is computed as a function of the following information:

- distributed application component attributes and interaction patterns between components;
- processing environment architecture and component reliabilities;
- the allocation of application components to processors in the processing environment.

The Reliability Analyzer is an integrated tool. Under user guidance it accesses the IP for its input and output data requirements. This facilitates the process of defining reliability analysis parameters and experimenting with alternative application decompositions in the process of building a distributed application that meets reliability objectives.

5.3.2.3. Implementation Notes

The Reliability Analyzer employs a three phase algorithm for computing availabilities. First, an annotated call graph is constructed and analyzed for dependencies between software components. Then, on the basis of the allocation of those components, hardware platform dependencies are computed. Finally, using reliabilities of hardware components, the availability can be computed for the target software component.

The Reliability Analyzer is a Cronus application written in C. It accesses the IP through Reliability Analyzer's IP Cronus type manager. The tool's user interface is developed using SunView, and it executes on Sun workstations.

5.4. Summary

This section has motivated the need for development tools, defined the purposes they serve and presented overviews of the Allocator and Reliability Analyzer tools.

Developers require high-technology tools to support the process of constructing efficient, reliable and correct distributed applications. The Allocator and Reliability Analyzer tool exemplify this type of technology by addressing, at design time, the challenges of building efficient and reliable distributed applications. In addition they provide complete illustrations of integrated tools and the benefits realized in using integrated tools.

SECTION 6

Conclusions and Future Directions

6.1. Summary

This report has summarized the results of the DOS Design Application Tools contract. A Tool Integration Platform was constructed that serves as an important element of an integrating development framework. It integrates tools by coordinating the interactions between them and by standardizing the entities that comprise distributed application development in DISE. Two development tools have been designed and implemented that illustrate the type of technology needed by distributed application developers and that provide a rigorous demonstration of the Integration Platform.

6.2. Future Directions

DISE is an evolving computing environment for developing, executing and evaluating distributed systems technologies. It contains a diverse and evolving set of development tools. DISE's mission therefore demands integration technology that allows components in the computing environment to work together cohesively. The IP provides a cornerstone for an integration framework upon which other integration mechanisms can be built. There are a number of directions for future work in the area of integration framework technologies.

6.2.1. Integration of Tools

An tool integration mechanism's utility is directly proportional to the number of tools integrated into it. An important area for future work is therefore in the area of integrating additional tools into the IP. This includes new tools currently, or soon to be, under development as well as previously existing tools.

The process of integrating tools is generally simpler if it is done while the tool is being developed. As new tools are developed and targeted to DISE, their integration should therefore be considered up-front rather than as an afterthought. As detailed elsewhere, this requires that the tool developer: 1) map the new tool's I/O data into the IP's information model; 2) develop a tool object manager; 3) insert, in the tool, calls to the tool's object manager; and 4) provide some means in the tool's interface for governing access to the IP in an appropriate fashion.

There are a wide variety of existing tools for which integration would be beneficial. Standard development tools such as editors and compilers need to be integrated. In the near term, it is important to integrate tools provided by the Cronus distributed operating system (such as *genmgr*). Much of the IP's information model is geared towards capturing the byproducts of these tools.

Integrating existing tools, however, is more difficult than integrating a new tool since it is typically not possible to change the tool's source code (due to proprietary considerations). And even if it is possible, it may well make for a difficult modification since the code would have to be well-documented to have any confidence at all that errors are not being introduced into the code. Nevertheless, if source code is available, making modifications to it may provide the best route to integration.

In cases where source code is not available, the approach would be to build tool adapters. A tool adapter is software that implements an interface between the existing tool and the IP. On the one side it would interact with the tool's existing interface while on the other it would interact with an object manager (just like any other tool does). For batch oriented tools (such as most compilers) where I/O occurs via the file system and command line parameters, adapters are fairly easy to build. The parameters and the contents of files named in the command line can be converted by the adapter into data items that match data in the IP.

Some existing tools, on the other hand, have highly sophisticated user interfaces. Intercepting I/O activity under these circumstances can be quite difficult -- it would require an equally sophisticated adapter sitting between the tool and the resident operating system. Alternatively, an off-line approach could be employed using pre- and post-processing. The adapter would be executed before and after execution of the tool to handle all IP accesses. The tool user would be responsible for coordinating adapter execution.

6.2.2. Automatic Tool Adapter Generation

Generalizing on the notion of adapters discussed above, a utility could be designed and developed to automate the construction of tool adapters. Such tools would take as input a specification of the tool's I/O requirements and, using a schema integration algorithm, would automatically generate the code for converting data entities between the tool and the IP. The key task here would be to define a formal specification technique for defining tool I/O in a precise and meaningful fashion. Such a tool might have a high payoff in an environment that hosts numerous foreign tools.

6.2.3. Independent View Mechanisms

The IP defines a single information model for use by all tools. The translation rules employed by a particular tool, on the other hand, define a tool-specific view of the information model. Since each tool employs an object manager for interfacing to the IP, the tool object managers can themselves be considered to be view mechanisms. As tools are added to the development framework, the possibility of tools sharing object managers (i.e., sharing precisely the same view of the data in the IP) becomes increasingly likely. The *independent view mechanisms* would provide a particular view of the IP's data to which any tool could subscribe. Investigations are needed to determine how this impacts the integration of tool sets and what additional functionality these object managers might be able to offer.

6.2.4. Information Model Extensions

The information model provided by the IP captures a large amount of distributed application development data, but by no means does it encompass all possible data that might need to be captured. The information model will naturally have to evolve along with the evolution of the DISE tool set. In fact, the capability to easily extend it was one of the primary factors influencing the selection of an object-oriented database substrate. Extending the information model so that it contains more entities and relationships will result in a corresponding extension to the set of tools it can readily serve.

6.2.5. Version Control and Configuration Management Functions

Configuration management deals with the components of software under development; version control deals with component versions. A configuration comprises particular versions of each component in the product. A product family has different configurations to meet different needs. For example, an application's development configuration might include an editor, debugger and instrumentation monitor while the production configuration would not have those components. The same configuration might have a stable version comprising only released versions of each component and an experimental version containing some pre-release components.

Having a functional IP makes it possible to insert version control functions directly in the IP, as the repository of record. Every object in the IP that can have versions can contain versioning methods. The methods will store a new version subject to some version determination policy, and deliver appropriate versions for IP access operations. The present IP schema contains the objects program family, program model, and program model element that could eventually contain methods to provide configuration management. The functionality of an externally developed configuration management and version control tool can be embedded in an object oriented IP by through the development of the appropriate methods on certain object types.

6.2.6. Support Tools

Any database benefits from the introduction of support tools. The process of integrating tools, modifying the information model and schema, editing object instances in the database, and so on are activities that can all be made simpler through the use of database utilities such as ad-hoc query facilities, browsing tools and schema editors. Investigations are needed to determine the highest payoff utilities in the context of distributed application development in DISE.